版权相关注意事项：
1、书籍版权归著者和出版社所有
2、本PDF来自于各个广泛的信息平台，经过整理而成
3、本PDF仅限用于非商业用途或者个人交流研究学习使用
4、本PDF获得者不得在互联网上以任何目的进行传播
5、如果觉得书籍内容很赞，请一定购买正版实体书，多多支持编写高质量的图书的作者和相应的出版社！当然，如果图书内容不堪入目，质量低下，你也可以选择狠狠滴撕裂本PDF
6、技术类书籍是拿来获取知识的，不是拿来收藏的，你得到了书籍不意味着你得到了知识，所以请不要得到书籍后就觉得沾沾自喜，要经常翻阅！！经常翻阅
7、请于下载PDF后24小时内研究使用并删掉本PDF

仅供非商业用途或交流学习使用

Broadview®
www.broadview.com.cn

etcd 技术内幕

百里燊 / 编著



中国工信出版集团
電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

作者简介

百里燊

硕士研究生毕业，小时候想成为闯荡江湖的侠客，结果着迷于各种代码，最终沦为辛勤工作的程序员。期待与大家进行交流。

邮箱：shen_baili@163.com

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

etcd 技术内幕

百里燊 / 编著



電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING



仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

内 容 简 介

etcd 是一个可靠的分布式 KV 存储产品，由 CoreOS 公司开发，其底层使用 Raft 算法保证一致性，主要用于共享配置和服务发现。

本书主要从源码角度深入剖析 etcd，首先介绍 etcd 的背景知识，如何搭建源码环境及相关的命令。接着从基本的 Raft 协议开始介绍，帮助读者了解 Raft 协议的背景、如何处理各种异常场景及相关扩展。然后分析 etcd-raft 模块对 Raft 协议的实现，同时介绍 etcd 自带的 raftexample 示例，帮助读者了解 etcd-raft 模块的基本使用方法。本书还介绍 HTTP 编程基础和 etcd-rafthttp 模块的工作原理及具体实现，以及 etcd 中如何处理 WAL 日志文件及快照数据文件，并且详细分析 etcd 的底层存储，对 v2 版本和 v3 版本的存储都做了详细的介绍。最后重点介绍 etcd 服务端和客户端的相关内容，etcd 服务端会组装并协调前面介绍的各个组件，并且在它们的基础上扩展出了更多的功能，此外还详细分析 v2 和 v3 两个版本客户端的具体实现。

本书适合 Go 语言开发者，以及对 etcd 技术感兴趣的读者阅读。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。
版权所有，侵权必究。

图书在版编目（CIP）数据

etcd 技术内幕 / 百里燊编著. —北京：电子工业出版社，2018.7
ISBN 978-7-121-34386-5

Ⅰ. ①e… Ⅱ. ①百… Ⅲ. ①分布式存贮器 Ⅳ. ① TP333.2

中国版本图书馆 CIP 数据核字（2018）第 122913 号

责任编辑：陈晓猛
印　　刷：三河市华成印务有限公司
装　　订：三河市华成印务有限公司
出版发行：电子工业出版社
　　北京市海淀区万寿路 173 信箱　　邮编：100036
开　　本：787×980　1/16　　印张：25.75　　字数：490 千字
版　　次：2018 年 7 月第 1 版
印　　次：2018 年 7 月第 1 次印刷
定　　价：89.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式：010-51260888-819，faq@phei.com.cn。



仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

前　言

etcd 是一个可靠的分布式 KV 存储，其底层使用 Raft 算法保证一致性，主要用于共享配置和服务发现。etcd 是 CoreOS 公司发起的一个开源项目，其源代码地址为 https://github.com/coreos/etcd。

目前提供配置共享和服务发现功能的组件还是比较多的，其中应用最广泛、大家最熟悉的应该就是 ZooKeeper 了，很多开源项目都在不同程度上依赖了 ZooKeeper，例如，Dubbo、Kafka。在 Golang 社区中，etcd 是唯一一个可以媲美 ZooKeeper 的组件，在有些方面，etcd 甚至超越了 ZooKeeper，给开发者眼前一亮的感觉。

etcd 作为一个优秀的分布式 KV 存储产品，其底层的 etcd-raft 模块实现了 Raft 协议，可以帮助开发者快速实现最终一致性功能。etcd 以其高性能、易维护、Raft 实现等优点，受到越来越多的开发人员的青睐，在 Golang 社区中声名大噪。

etcd 的代码中有很多亮点，为了提高整体性能，其实现中使用了大量的 goroutine 和 Channel。etcd 3 中开始使用 BoltDB 作为底层的持久存储，使用 BTree 索引加速查询，还提供了可靠的 Watcher 功能，同时提供了基于 GRPC 的新版本客户端。当然，etcd 3 依然兼容 etcd 2 的内存存储和 HTTP API。

etcd 3 中有很多令人称赞的功能和优秀的设计，但至今还没有深入剖析 etcd 3 的内部设计和实现细节的图书，本书以 etcd 3 为基础，详细分析了 etcd 的架构设计和实现细节，其中穿插介绍了 etcd 源码中涉及的基础知识，以及笔者在实践中的思考。

如何阅读本书

由于篇幅限制，本书并没有详细介绍 Go 语言的基础知识，但为了便于理解读者理解 etcd 的设计思想和实现细节，笔者介绍了一些必需且重要的基础内容，例如，Go 语言提供的 HTTP 功能。

本书共 8 章，主要从源码角度深入剖析 etcd 的原理和实现。建议读者先阅读前两章，了解



仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

Raft 协议之后，再开始学习 etcd 的工作原理和代码实现。

第 1 章简要介绍 etcd 的背景知识及其基本的数据模型，然后介绍如何搭建源码环境及相关的命令。

第 2 章从基本的 Raft 协议开始介绍，帮助读者了解 Raft 协议的背景知识、大致工作原理、如何处理各种异常场景，以及几个比较重要的扩展点。

第 3 章着重分析 etcd 中的 raft 模块对 Raft 协议的实现，etcd-raft 模块是 etcd 的核心模块之一，GitHub 上很多其他项目直接使用 etcd-raft 模块作为其 Raft 协议的底层实现。同时，本章也介绍了 etcd 自带的 raftexample 示例，帮助读者了解 etcd-raft 模块的基本使用方法，方便读者在实践中直接使用 etcd-raft 模块。

第 4 章简单介绍 HTTP 编程基础，以及 etcd-rafthttp 模块的工作原理和具体实现，其中涉及 RoundTripper 的基本原理、rafthttp.Transporter 的实现及多种 Handler 的实现。

第 5 章介绍 etcd 中如何处理 WAL 日志文件和快照数据文件，其中分析了 WAL 结构体和 Snapshotter 的具体实现。

第 6 章详细分析 etcd 的底层存储，对 etcd v2 和 etcd v3 两个版本的存储都做了详细的介绍，不仅分析了底层的 KV 存储，还分析了 watcher 机制和 Lessor 的实现原理。

第 7 章重点介绍 etcd 服务端的内容，etcd 服务端会组装并协调前面介绍的各个组件，并且在它们的基础上扩展出了更多的功能。

第 8 章主要介绍 etcd 客户端的相关内容，详细分析了 v2 和 v3 两个版本客户端的具体实现。

如果读者在阅读本书的过程中，发现任何不妥之处，请将您宝贵的意见和建议发送到邮箱 shen_baili @163.com，也欢迎读者朋友通过此邮箱与我进行交流。

致谢

感谢我的母亲，谢谢您的付出和牺牲！

感谢白酱陪我看过一盏盏亮起的路灯，感谢三十在技术上提供的帮助。

感谢电子工业出版社博文视点的陈晓猛老师，是您的辛勤工作让本书的出版成为可能。同时还要感谢许多我不知道名字的幕后工作人员为本书付出的努力。

百里燊



仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

读者服务

轻松注册成为博文视点社区用户（www.broadview.com.cn），扫码直达本书页面。

- **提交勘误**：您对书中内容的修改意见可在提交勘误处提交，若被采纳，将获赠博文视点社区积分（在您购买电子书时，积分可用来抵扣相应金额）。
- **交流互动**：在页面下方读者评论处留下您的疑问或观点，与我们和其他读者一同学习交流。

页面入口：http://www.broadview.com.cn/34386





仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

目　录

第 1 章　etcd 入门 1
1.1　etcd 简介 1
1.2　数据模型 2
1.3　环境搭建 3
1.3.1　环境变量 3
1.3.2　代码结构 3
1.3.3　运行 4
本章小结 6

第 2 章　Raft 协议 7
2.1　Leader 选举 8
2.2　日志复制 15
2.3　网络分区的场景 21
2.4　日志压缩与快照 27
2.5　其他技术点 29
2.5.1　linearizable 语义 29
2.5.2　只读请求 29
2.5.3　PreVote 状态 30
2.5.4　Leader 节点转移 31
本章小结 31

第 3 章　etcd-raft 模块详解 33
3.1　raft 结构体 35
3.1.1　Config 结构体 37



仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

3.1.2 Storage 接口及其实现 38
3.1.3 unstable 结构体 43
3.1.4 raftLog 结构体 48
3.1.5 raft 实现 55
3.2 Node 接口 95
3.2.1 node 结构体 98
3.2.2 初始化 98
3.2.3 run()方法 100
3.2.4 Node 接口实现 109
3.3 raftexample 示例分析 110
3.3.1 raftNode 111
3.3.2 HTTP 服务端 125
3.3.3 kvstore 128
本章小结 129

第 4 章 网络层 130
4.1 Go 语言网络编程基础 130
4.1.1 http.Server 130
4.1.2 RoundTripper 135
4.2 etcd-rafthttp 模块详解 142
4.2.1 rafthttp.Transporter 接口 144
4.2.2 Peer 接口 148
4.2.3 pipeline 152
4.2.4 streamWriter 实例 155
4.2.5 streamReader 实例 159
4.2.6 snapshotSender 162
4.3 Handler 实例 164
4.3.1 pipelineHandler 164
4.3.2 streamHandler 165
4.3.3 snapshotHandler 166
本章小结 167

第 5 章 WAL 日志与快照 169
5.1 WAL 日志 169



仅供非商业用途或交流学习使用

5.1.1 初始化 174
5.1.2 打开日志 176
5.1.3 读取日志 177
5.1.4 追加日志 181
5.1.5 文件切换 184
5.2 SnapShotter 186
本章小结 189

第 6 章 storage 190

6.1 etcd v2 版本存储 190
6.1.1 node 192
6.1.2 Event 197
6.1.3 watcher 和 watcherHub 200
6.1.4 store 205
6.2 etcd v3 版本存储 216
6.2.1 backend 217
6.2.2 watcher 机制 274
6.2.3 Lessor 292
本章小结 300

第 7 章 etcd-server 详解 302

7.1 raftNode 结构体 302
7.2 RaftCluster 311
7.3 EtcdServer 312
7.3.1 初始化 315
7.3.2 注册 Handler 326
7.3.3 启动 328
7.4 ApplierV2 和 applierV3 350
7.4.1 ApplierV2 350
7.4.2 applierV3 353
7.5 AlarmStore 366
本章小结 369



仅供非商业用途或交流学习使用

第 8 章 etcd 客户端详解 371

8.1 GRPC 基础 371

8.1.1 定义 proto 文件 371

8.1.2 服务端 373

8.1.3 创建客户端 376

8.2 Client v3 376

8.2.1 kvServer 378

8.2.2 EtcdServer 379

8.2.3 Client 387

8.3 Client v2 394

8.3.1 KeysAPI 接口 394

8.3.2 httpClient 接口 396

本章小结 401

参考文献 402





第 1 章 etcd 入门

1.1 etcd简介

etcd 是一个可靠的分布式 KV 存储，其底层使用 Raft 算法保证一致性，主要用于共享配置和服务发现。etcd 是 CoreOS 公司发起的一个开源项目，授权协议为 Apache，其源代码地址为 https://github.com/coreos/etcd。

目前提供配置共享和服务发现功能的组件还是比较多的，其中应用最为广泛、大家最为熟悉的应该就是 ZooKeeper 了，很多开源项目也都在不同程度上依赖了 ZooKeeper，例如，Dubbo、Kafka 等。

这里简单介绍一下“服务发现”和“共享配置”两个概念。随着一个系统的不断迭代，功能模块会不断增加，对整个系统进行服务的拆分是必然的，这就会出现多个服务之间的相互调用，而在出现服务发现组件之前，一般是通过读取配置文件中预先设置的 IP 来获取服务的地址，然后进行调用。这会导致很多问题，例如，某些服务已经不可用时，调用方不能及时感知，负载均衡比较复杂，等等。使用服务发现组件之后，我们可以将服务提供方的信息注册到服务发现组件，例如，注册到 ZooKeeper 中，然后定期发送心跳等信息，让服务发现组件知晓服务提供方是可用的。当调用方进行服务调用时，会先请求服务发现组件，由服务发现组件来保证返回可用的服务地址及负载均衡等功能。

在一个系统的不同模块中有很多配置信息，例如，数据库地址、连接配置信息等都差不多，如果使用静态配置文件的方式实现，则需要将相同的信息写多份，每次更新配置时也需要更新多次。如果将这些配置信息注册到共享配置组件中，则系统的不同模块在启动时可从共享配置

组件中获取配置，同时会监听配置信息的更改，当配置信息发生变更时，可以自动将配置值替换成新值。

在 Golang 社区中，etcd 则是唯一一个可以媲美 ZooKeeper 的组件，在有些方面，etcd 甚至超越了 ZooKeeper，给开发者眼前一亮的感觉。下面简单列举一下 etcd 相较于 ZooKeeper 的优势。

- **一致性协议**：一致性协议是配置共享和服务发现组件的核心，etcd 底层采用了 Raft 协议，而 ZooKeeper 使用 ZAB 协议，ZAB 协议是一种类 Paxos 的一致性协议。目前公认的是 Raft 比 Paxos 协议易于理解，工程化也较为容易。
- **API 接口**：etcd v2 版本中提供了 HTTP+JSON 的调用方式，在 etcd v3 版本的客户端中则使用 GRPC 与服务端进行交互，而 GRPC 本身就是跨平台的。
- **性能**：在官方提供的基准测试数据中，etcd 集群可以支持每秒 10000+次的写入，性能相当可观，优于 ZooKeeper。
- **安全性**：etcd 支持 TLS 访问，而 ZooKeeper 在权限控制这方面做得略显粗糙。

etcd 也有很多成功的案例，最为大家熟知的就是 Kubernetes，其底层就依赖于 etcd 实现集群状态和配置的管理。除了 Kubernetes，Cloud Foundry 等 500+个 GitHub 项目都使用了 etcd。etcd 在 GitHub 上也有 17000+的 Star 和近 4000 的 Fork。

1.2 数据模型

etcd 支持可靠的键值对存储并且提供了可靠的 Watcher 机制，其中的键值对存储支持多版本，并且具备能够“Watch”历史事件的功能。这里简单介绍多版本存储的含义，假设键 K1 对应的值为 V1，当我们将 K1 对应的值修改成 V2 时，etcd 并不会直接将 V1 修改成 V2，而是同时记录 V1 和 V2 两个值，并通过不同的版本号进行区分。另外，Watch 历史事件的含义是，我们可以向一个 Key 添加 Watcher，同时可以指定一个历史版本，从该版本开始的所有事件都会触发该 Watcher。

随着应用不断运行，键值对不断修改，每个 Key 都在 etcd 中保存了多个版本，数据量也会越来越大。为了缓解压力，etcd 会定期进行压缩，清理过旧的数据。

在很多现代数据库系统中，都用了 B 树索引加速查询，etcd 也是如此，其存储中会维护一个字段序的 B 树索引。在 B 树索引的每个索引项中，都存储了一个 Key 值，这样可以快速定位指定的 Key 或是进行范围查询。而每个 Key 值对应了多个版本号，etcd 中维护了一个全局自增的版本号，为每次事务分配一个全局唯一的版本号（main revision），事务中的每个操作也有唯一的编号（sub revision），通过这两部分可以确定一个唯一的 Value 值。

每个Key会对应多个generation，当Key首次创建时，会同时创建一个与之关联的generation实例，当该Key被修改时，会将对应的版本记录到generation中，当Key被删除时，会向generation中添加tombstone，并创建新的generation，会向新generation中写入后续的版本信息。

在查询时，先在内存索引中通过用户指定的Key值，查找到该Key值对应的全部版本号，然后根据用户指定的版本号，从底层存储中查找到具体的Value值。当然，如果指定的版本号已经被etcd压缩删除，则无法再查询到该版本的Value值。

在etcd v3版本中，底层存储使用的是BoltDB，其中的Key是版本信息（main revision+sub revision）。这样，在查询时先通过上述B树索引查找到对应的版本信息，然后在BoltDB中通过版本信息查找相应的Value值。

1.3 环境搭建

了解了etcd中基本的概念和数据模型之后，我们开始搭建阅读etcd源码的环境。这里推荐使用Goland，如果读者熟悉其他IDE，也可以按照下面的步骤进行搭建。

1.3.1 环境变量

首先需要从Golang官方网站上下载最新的Golang包，笔者在开始写作时的最新版本是go1.10.darwin-amd64.tar.gz，下载完成后进行解压。接下来我们需要配置相应的环境变量，打开~/.bash_profile文件，添加如下内容：

```
export GOROOT=/Users/xxx/Documents/go
export GOBIN=$GOROOT/bin
export GOARCH=amd64
export GOOS=darwin
export GOPATH=/Users/xxx/Documents/GoProject
export PATH=$PATH:$GOBIN:$GOPATH/bin
```

如果读者使用了其他终端，则需要在相应的终端配置文件中添加上述配置。最后执行一下“source .bash_profile”命令编译配置文件。我们可以通过“go env”命令验证环境变量配置是否成功。

1.3.2 代码结构

我们可以直接使用“go get”命令下载etcd的源码代码，具体命令如下：

```
go get github.com/coreos/etcd/cmd/etcd
```

下载过程可能比较长，请读者耐心等待，待下载完成之后，即可以将代码直接导入 Goland 中了。这里简单介绍其中各个模块的主要功能，如下所示。

- raft：Raft 协议的核心实现，其中只实现了基本的 Raft 协议，并未提供实现网络层相关的内容。
- raft-http：Raft 协议中各个节点发送消息时使用的网络层实现，该模块与 raft 模块紧密相关。
- wal 和 snap：WAL 日志和快照存储相关的实现。
- store：etcd 中的 v2 版本存储实现，v2 版本的存储是完全的内存实现。
- mvcc：etcd 中的 v3 版本存储实现，v3 版本的存储底层使用 BoltDB 实现持久化存储。
- lease：租约相关的实现。
- auth 和 alarm：权限和报警相关的实现。
- etcdserver：etcd 服务端实现，它会基于上述模块提供的功能，对外提供一个 etcd 节点的完整功能。
- client：v2 版本客户端的具体实现，v2 版本的客户端是通过 HTTP+JSON 的方式与服务端进行交互的。
- clientv3：v3 版本客户端的具体实现，v3 版本的客户端是通过 GRPC 的方式与服务端进行交互的。

1.3.3 运行

etcd 提供了单机模式和集群模式两种模式，单机模式比较简单，直接编译之前下载的源代码得到/bin/etcd 二进制文件并运行即可。默认配置运行时，etcd 服务端会监听本地的 2379 和 2380 两个端口，其中 2379 端口用于与客户端的交互，2380 端口则是用于 etcd 节点内部交互（主要是发送 Raft 协议相关的消息等）。当 etcd 服务端启动时，我们可以使用 etcdctl 工具进行测试，关于 etcdctl 命令的使用，这里不再展开介绍，请读者参考官方文档进行学习。

本小节重点介绍集群模式的搭建。为了方便测试和部署，在本小节的示例中将会在同一台机器上启动三个 etcd 节点，但每个节点监听不同的端口，形成伪分布式的集群。

首先介绍静态配置的启动方式，这种方式会预先将集群中各个节点的配置信息分配好，然后将集群中的节点逐个启动，这些节点将根据配置信息组成集群。在该示例中，三个节点启动

的代码分别如下所示。

```
./bin/etcd --name infra0 --initial-advertise-peer-urls http://127.0.0.1:2380 \
  --listen-peer-urls http://127.0.0.1:2380 \
  --listen-client-urls http://127.0.0.1:2379 \
  --advertise-client-urls http://127.0.0.1:2379 \
  --initial-cluster-token etcd-cluster-1 \
  --initial-cluster infra0=http://127.0.0.1:2380, \
    infra1=http://127.0.0.1:2382,infra2=http://127.0.0.1:2384 \
  --initial-cluster-state new

./bin/etcd --name infra1 --initial-advertise-peer-urls http://127.0.0.1:2382 \
  --listen-peer-urls http://127.0.0.1:2382 \
  --listen-client-urls http://127.0.0.1:2381 \
  --advertise-client-urls http://127.0.0.1:2381 \
  --initial-cluster-token etcd-cluster-1 \
  --initial-cluster infra0=http://127.0.0.1:2380,\
    infra1=http://127.0.0.1:2382,infra2=http://127.0.0.1:2384 \
  --initial-cluster-state new

./bin/etcd --name infra2 --initial-advertise-peer-urls http://127.0.0.1:2384 \
  --listen-peer-urls http://127.0.0.1:2384 \
  --listen-client-urls http://127.0.0.1:2383 \
  --advertise-client-urls http://127.0.0.1:2383 \
  --initial-cluster-token etcd-cluster-1 \
  --initial-cluster infra0=http://127.0.0.1:2380,\
    infra1=http://127.0.0.1:2382,infra2=http://127.0.0.1:2384 \
  --initial-cluster-state new
```

这里简单介绍一下上述命令使用的参数。

- --name：etcd 集群中的节点名称，在同一个集群中必须是唯一的。
- --listen-peer-urls：用于集群内各个节点之间通信的 URL 地址，每个节点可以监听多个 URL 地址，集群内部将通过这些 URL 地址进行数据交互，例如，Leader 节点的选举、Message 消息传输或是快照传输等。
- --initial-advertise-peer-urls：建议用于集群内部节点之间交互的 URL 地址，节点间将以该值进行通信。

- --listen-client-urls：用于当前节点与客户端交互的 URL 地址，每个节点同样可以向客户端提供多个 URL 地址。
- --advertise-client-urls：建议客户端使用的 URL 地址，该值用于 etcd 代理或 etcd 成员与 etcd 节点通信。
- --initial-cluster-token etcd-cluster-1：集群的唯一标识。
- --initial-cluster：集群中所有的 initial-advertise-peer-urls 的合集。
- --initial-cluster-state new：新建集群的标识。

当集群中各个节点按照上述配置分别启动后，集群会通过 Leader 选举选出一个 Leader 节点，另外两个节点将成为 Follower 节点。我们可以使用 etcdctl 命令检测当前集群的状态，etcdctl 的相关使用方式请读者参考官方文档进行学习。

除了静态配置的启动方式，etcd 还提供了服务发现和 DNS 发现两种启动方式，这两种启动方式并不复杂，这里不再一一展开介绍，读者可以参考官方文档进行测试。

本章小结

本章首先对 etcd 进行了简单的介绍，以及介绍 etcd 相较于 ZooKeeper 的优点。之后，我们简单介绍了 etcd 的数据模型，其中提到 etcd 中在内存中维护了 B 树索引，并且为每个 Key 维护了多个版本，在 etcd v3 中使用 BoltDB 作为持久存储，而 etcd v2 则是全内存实现。然后，我们简单介绍了 etcd 源码环境的搭建及代码结构，最后介绍了 etcd 集群中最基本的静态配置启动方式。希望通过本章的介绍，读者可以大致了解 etcd 的数据结构及代码结构，也希望读者能够自己亲自完成 etcd 的源码环境搭建，并使用静态配置的方式启动本地的测试集群。



第 2 章 Raft 协议

在正式开始介绍 Raft 协议之间，我们有必要简单介绍一下其相关概念。在分布式系统中，一致性是比较常见的概念，所谓一致性指的是集群中的多个节点在状态上达成一致。在程序和操作系统不会崩溃、硬件不会损坏、服务器不会掉电、网络绝对可靠且没有延迟的理想情况下，我们可以将集群中的多个节点看作一个整体，此时要保证它们的一致性并不困难。

但是在现实的场景中，很难保证上述极端的条件全部满足，节点之间的一致性也就很难保证，这样就需要 Paxos、Raft 等一致性协议。一致性协议可以保证在集群中**大部分节点可用**的情况下，集群依然可以工作并给出一个正确的结果，从而保证依赖于该集群的其他服务不受影响。这里的“**大部分节点可用**”指的是集群中超过半数以上的节点可用，例如，集群中共有 5 个节点，此时其中有 2 个节点出现故障宕机，剩余的可用节点数为 3，此时，集群中大多数节点处于可用的状态，从外部来看集群依然是可用的。

常见的一致性算法有 Paxos、Raft 等，Paxos 协议是 Leslie Lamport 于 1990 年提出的一种基于消息传递的、具有高度容错特性的一致性算法，Paxos 算法解决的主要问题是分布式系统内如何就某个值达成一致。在相当长的一段时间内，Paxos 算法几乎成为一致性算法的代名词，但是 Paxos 有两个明显的缺点：第一个也是最明显的缺点就是 Paxos 算法难以理解，Paxos 算法的论文本身就比较晦涩难懂，要完全理解 Paxos 协议需要付出较大的努力，很多经验丰富的开发者在看完 Paxos 论文之后，无法将其有效地应用到具体工程实践中，这明显增加了工程化的门槛，也正因如此，才出现了几次用更简单的术语来解释 Paxos 的尝试。Paxos 算法的第二个缺点就是它没有提供构建现实系统的良好基础，也有很多工程化 Paxos 算法的尝试，但是它们对 Paxos 算法本身做了比较大的改动，彼此之间的实现差距都比较大，实现的功能和目的都有所不同，同时与 Paxos 算法的描述有很多出入。例如，著名 Chubby，它实现了一个类 Paxos 的算法，但其中很多细节并未被明确。本章并不打算详细介绍 Paxos 协议的相关内容，如果读

者对 Paxos 感兴趣，则可以参考 Lamport 发表的三篇论文：《*The Part-Time Parliament*》、《*Paxos made simple*》、《*Fast Paxos*》。

正因为上述的缺点，导致 Paxos 协议处于一种比较尴尬的境地：在理论上 Paxos 算法是正确可行的，但是实际的工程中很少有与 Paxos 算法类似的实践。很多工程实践（包括上面提到的 Chubby）都是从 Paxos 协议的研究开始的，然后在实践的过程中发现很多难题，之后通过各种技巧和手段进行改进，最后开发出一种与 Paxos 明显不同的东西，这就导致最终开发出来的程序建立在一个未经证明的协议之上。也正因为如此，人们开始寻找新的一致性算法，寻找的结果也就是本章介绍的重点——Raft 协议。

Raft 算法是一种用于管理复制日志的一致性算法，其功能与 Paxos 算法相同类似，但其算法结构和 Paxos 算法不同，在设计 Raft 算法时设计者就将易于理解作为其目标之一，这使得 Raft 算法更易于构建实际的系统，大幅度减少了工程化的工作量，也方便开发者此基础上进行扩展。虽然 Raft 论文已经很好理解，但是本章并不打算直接翻译 Raft 论文，而是尽可能通过示例介绍 Raft 协议如何处理各种不同的场景，并且重点介绍 Raft 协议中的 Leader 选举和日志复制等方面的内容。

2.1 Leader选举

Raft 协议的工作模式是一个 Leader 节点和多个 Follower 节点的模式，也就是常说的 Leader-Follower 模式。在 Raft 协议中，每个节点都维护了一个状态机，该状态机有三种状态，分别是 Leader 状态、Follower 状态和 Candidate 状态，在任意时刻，集群中的任意一个节点都处于这三个状态之一。各个状态和转换条件如图 2-1 所示。

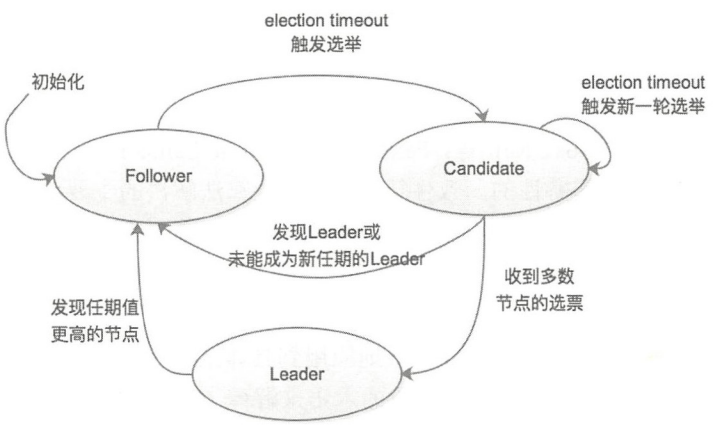


图 2-1

在多数情况下，集群中有一个 Leader 节点，其他节点都处于 Follower 状态，下面简单介绍一下每个状态的节点负责的主要工作。

- Leader 节点负责处理所有客户端的请求，当接收到客户端的写入请求时，Leader 节点会在本地追加一条相应的日志，然后将其封装成消息发送到集群中其他的 Follower 节点。当 Follower 节点收到该消息时会对其进行响应。如果集群中多数（超过半数）节点都已收到该请求对应的日志记录时，则 Leader 节点认为该条日志记录已提交（committed），可以向客户端返回响应。Leader 还会处理客户端的只读请求，其中涉及一个简单的优化，后面介绍具体实现时，再进行详细介绍。Leader 节点的另一项工作是定期向集群中的 Follower 节点发送心跳消息，这主要是为了防止集群中的其他 Follower 节点的选举计时器超时而触发新一轮选举。
- Follower 节点不会发送任何请求，它们只是简单地响应来自 Leader 或者 Candidate 的请求；Follower 节点也不处理 Client 的请求，而是将请求重定向给集群的 Leader 节点进行处理。
- Candidate 节点是由 Follower 节点转换而来的，当 Follower 节点长时间没有收到 Leader 节点发送的心跳消息时，则该节点的选举计时器就会过期，同时会将自身状态转换成 Candidate，发起新一轮选举。选举的具体过程在下面详细描述。

了解了 Raft 协议中节点的三种状态及各个状态下节点的主要行为之后，我们通过一个示例介绍 Raft 协议中 Leader 选举的大致流程。为了方便描述，我们假设当前集群中有三个节点（A、B、C），如图 2-2 所示。

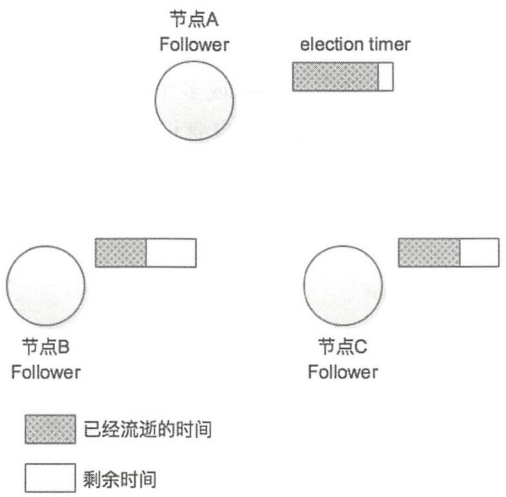


图 2-2

在 Raft 协议中有两个时间控制 Leader 选举发生，其中一个是选举超时时间（election timeout），每个 Follower 节点在接收不到 Leader 节点的心跳消息之后，并不会立即发起新一轮选举，而是需要等待一段时间之后才切换成 Candidate 状态发起新一轮选举。这段等待时长就是这里所说的

election timeout（后面介绍 etcd 的具体实现时会提到，Follower 节点等待的时长并不完全等于该配置）。之所以这样设计，主要是 Leader 节点发送的心跳消息可能因为瞬间的网络延迟或程序瞬间的卡顿而迟到（或是丢失），因此就触发新一轮选举是没有必要的。election timeout 一般设置为 150ms～300ms 之间的随机数。另一个超时时间是心跳超时时间（heartbeat timeout），也就是 Leader 节点向集群中其他 Follower 节点发送心跳消息的时间间隔。

当集群初始化时，所有节点都处于 Follower 的状态，此时的集群中没有 Leader 节点。当 Follower 节点一段时间（选举计时器超时）内收不到 Leader 节点的心跳消息，则认为 Leader 节点出现故障导致其任期（Term）过期，Follower 节点会转换成 Candidate 状态，发起新一轮的选举。所谓 “任期（Term）”，实际上就是一个全局的、连续递增的整数，在 Raft 协议中每进行一次选举，任期（Term）加一，在每个节点中都会记录当前的任期值（currentTerm）。每一个任期都是从一次选举开始的，在选举时，会出现一个或者多个 Candidate 节点尝试成为 Leader 节点，如果其中一个 Candidate 节点赢得选举，则该节点就会切换为 Leader 状态并成为该任期的 Leader 节点，直到该任期结束。

回到前面的示例中，此时节点 A 由于长时间未收到 Leader 的心跳消息，就会切换成为 Candidate 状态并发起选举（节点 A 的选举计时器（election timer）已被重置）。在选举过程中，节点 A 首先会将自己的选票投给自己，并会向集群中其他节点发送选举请求（Request Vote）以获取其选票，如图 2-3（1）所示；此时的节点 B 和节点 C 还都是处于 Term=0 的任期之中，且都是 Follower 状态，均未投出 Term=1 任期中的选票，所以节点 B 和节点 C 在接收到节点 A 的选举请求后会将选票投给节点 A，另外，节点 B、C 在收到节点 A 的选举请求的同时会将选举定时器重置，这是为了防止一个任期中同时出现多个 Candidate 节点，导致选举失败，如图 2-3（2）所示。注意，节点 B 和节点 C 也会递增自身记录的 Term 值。

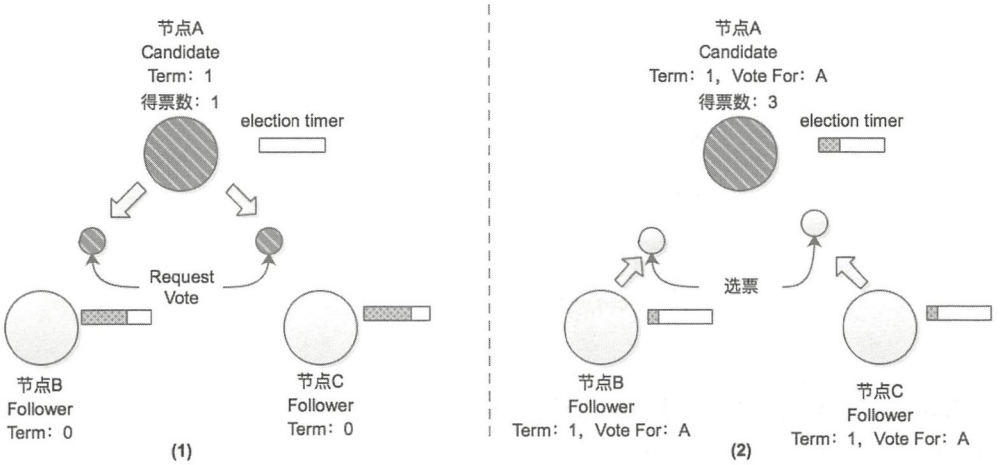


图 2-3

在节点 A 收到节点 B、C 的投票之后，其收到了集群中超过半数的选票，所以在 Term=1 这个任期中，该集群的 Leader 节点就是节点 A，其他节点将切换成 Follower 状态，如图 2-4 所示。另外需要读者了解的是，集群中的节点除了记录当期任期号（currentTerm），还会记录在该任期中当前节点的投票结果（VoteFor）。

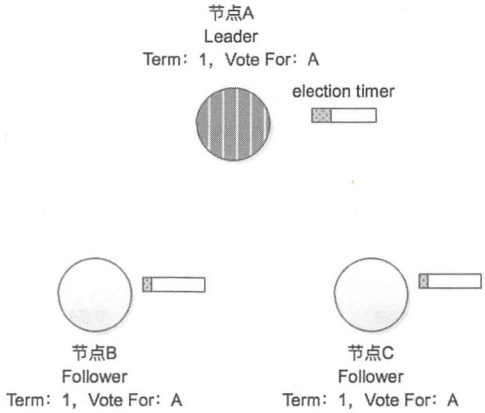


图 2-4

继续前面的示例，成为 Term=1 任期的 Leader 节点之后，节点 A 会定期向集群中的其他节点发送心跳消息，如图 2-5（1）所示，这样就可以防止节点 B 和节点 C 中的选举计时器（election timer）超时而触发新一轮的选举；当节点 B 和节点 C（Follower）收到节点 A 的心跳消息之后会重置选举计时器，如图 2-5（2）所示，由此可见，心跳超时时间（heartbeat timeout）需要远远小于选举超时时间（election timeout）。

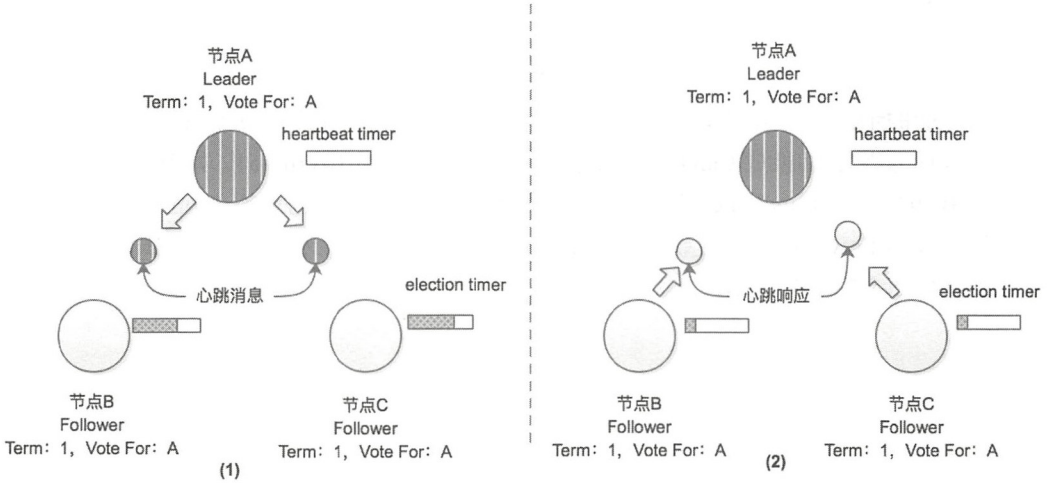


图 2-5

到这里读者可能会问，如果有两个或两个以上节点的选举计时器同时过期，则这些节点会同时由 Follower 状态切换成 Candidate 状态，然后同时触发新一轮选举，在该轮选举中，每个 Candidate 节点获取的选票都不到半数，无法选举出 Leader 节点，那么 Raft 协议会如何处理呢？这种情况确实存在，假设集群中有 4 个节点，其中节点 A 和节点 B 的选举计时器同时到期，切换到 Candidate 状态并向集群中其他节点发出选举请求，如图 2-6（1）所示。

这里假设节点 A 发出的选举请求先抵达节点 C，节点 B 发出的选举请求先抵达节点 D，如图 2-6（2）所示，节点 A 和节点 B 除了得到自身的选票之外，还分别得到了节点 C 和节点 D 投出的选票，得票数都是 2，都没有超过半数。在这种情况下，Term=4 这个任期会以选举失败结束，随着时间的流逝，当任意节点的选举计时器到期之后，会再次发起新一轮的选举。前面提到过 election timeout 是在一个时间区间内取的随机数，所以在配置合理的时候，像上述情况多次出现的概率并不大。

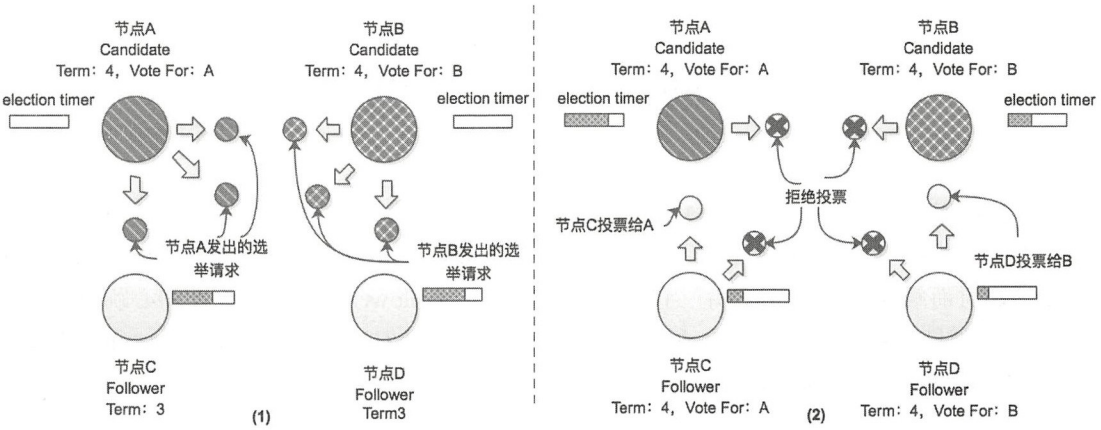


图 2-6

继续上面的示例，这里假设节点 A 的选举计时器再次到期（此次节点 B、C、D 的选举计时器并未到期），它会切换成 Candidate 状态并发起新一轮选举（Term=5），如图 2-7（1）所示，其中节点 B 虽然处于 Candidate 状态，但是接收到 Term 值比自身记录的 Term 值大的请求时，节点会切换成 Follower 状态并更新自身记录的 Term 值，所以该示例中的节点 B 也会将选票投给节点 A，如图 2-7（2）所示。

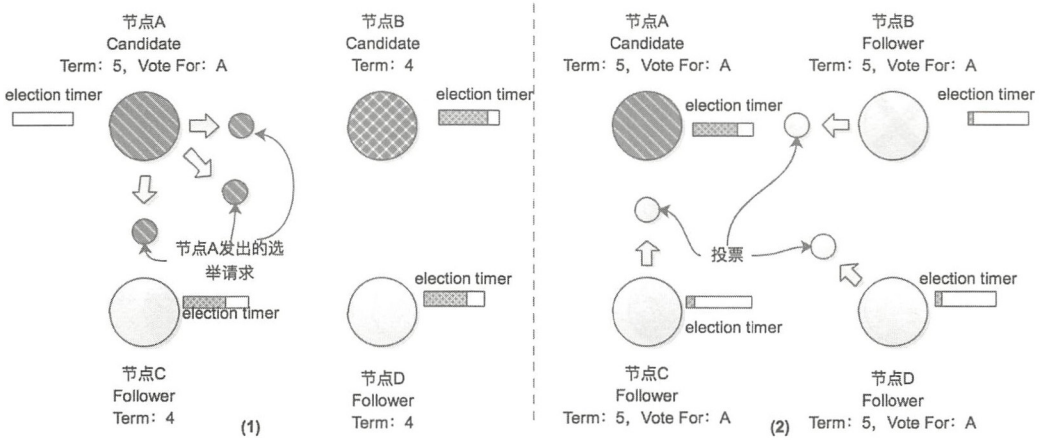


图 2-7

在获取集群中半数以上的选票并成为新任期（Term=5）的 Leader 之后，节点 A 会定期向集群中其他节点发送心跳消息；当集群中其他节点收到 Leader 节点的心跳消息的时候，会重置选举定时器，如图 2-8 所示。

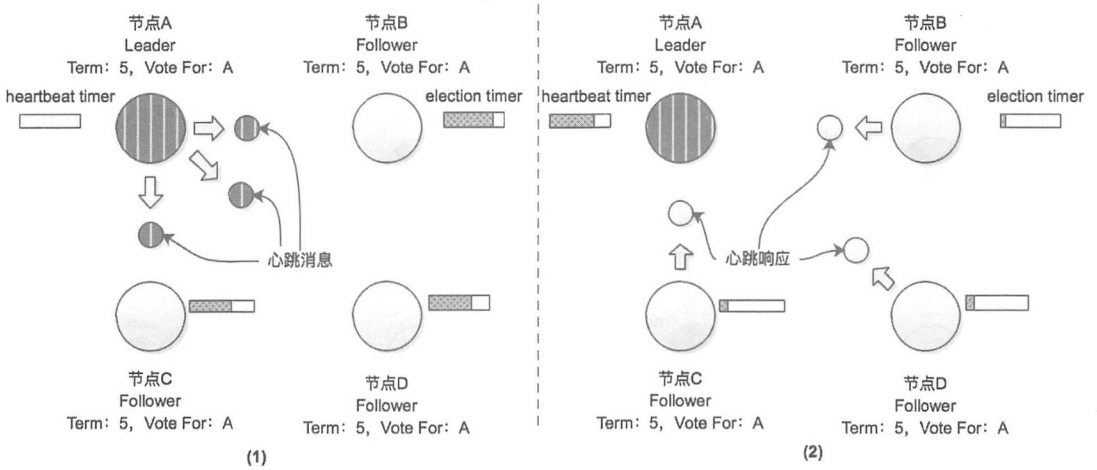


图 2-8

介绍完集群启动时的Leader选举流程之后，下面分析Leader节点宕机之后重新选举的场景。继续上述 4 节点集群的示例，在系统运行一段时间后，集群当前的 Leader 节点（A）因为故障而宕机，此时将不再有心跳消息发送到集群的其他 Follower 节点（节点 B、C、D），一段时间后，会有一个 Follower 节点的选举计时器最先超时，这里假设节点 D 的选举计时器最先超时，然后它将切换为 Candidate 状态并发起新一轮选举，如图 2-9（1）所示。

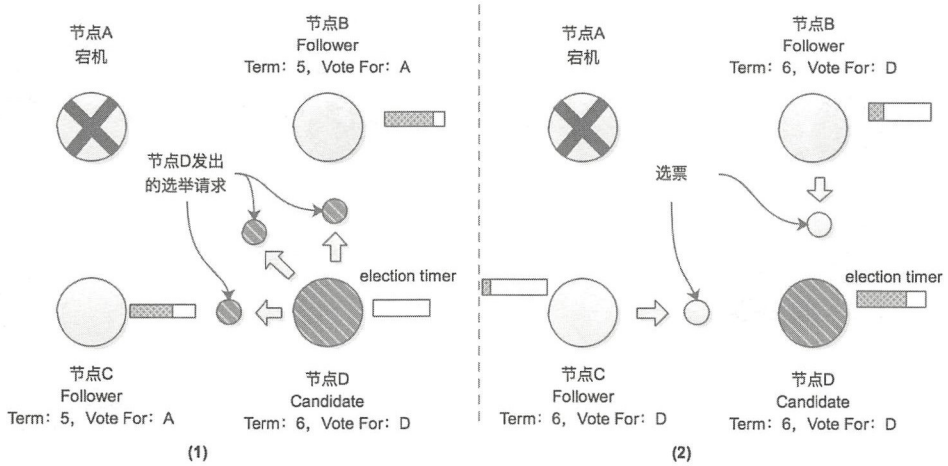


图 2-9

当节点 B 和节点 C 收到节点 D 的选举请求后，会将其选票投给节点 D，由于节点 A 已经宕机，没有参加此次选举，也就无法进行投票，但是在此轮选举中，节点 D 依然获得了半数以上的选票，故成为新任期（Term=6）的 Leader 节点，并开始向其他 Follower 节点发送心跳消息，如图 2-10 所示。

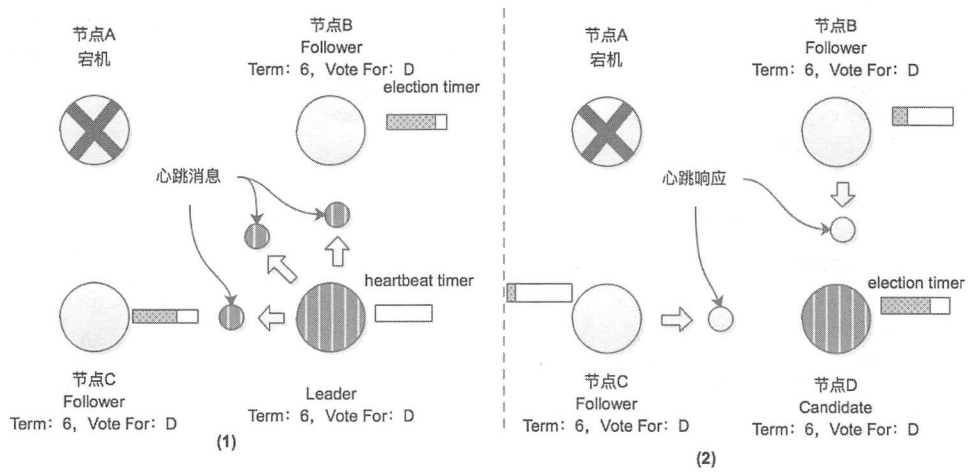


图 2-10

当节点 A 恢复之后，会收到节点 D 发来的心跳消息，该消息中携带的任期号（Term=6）大于节点 A 当前记录的任期号（Term=5），所以节点 A 会切换成 Follower 状态。在 Raft 协议中，当某个节点接收到的消息所携带的任期号大于当前节点本身记录的任期号，那么该节点会更新自身记录的任期号，同时会切换为 Follower 状态并重置选举计时器，这是 Raft 算法中所有节点

都要遵守的一条准则。

最后请读者考虑一个场景：如果集群中选出的 Leader 节点频繁崩溃或是其他原因导致选举频繁发生，这会使整个集群中没有一个稳定的 Leader 节点，这样客户端无法与集群中的 Leader 节点正常交互，也就会导致整个集群无法正常工作。

Leader 选举是 Raft 算法中对时间要求较为严格的一个点，一般要求整个集群中的时间满足如下不等式：

$$广播时间 << 选举超时时间 << 平均故障间隔时间$$

在上述不等式中，广播时间指的是从一个节点发送心跳消息到集群中的其他节点并接收响应的平均时间；平均故障间隔时间就是对于一个节点而言，两次故障之间的平均时间。为了保证整个 Raft 集群可用，广播时间必须比选举超时时间小一个数量级，这样 Leader 节点才能够发送稳定的心跳消息来重置其他 Follower 节点的选举计时器，从而防止它们切换成 Candidate 状态，触发新一轮选举。在前面的描述中也提到过，选举超时时间是一个随机数，通过这种随机的方式，会使得多个 Candidate 节点瓜分选票的情况明显减少，也就减少了选举耗时。另外，选举超时时间应该比平均故障间隔时间小几个数量级，这样 Leader 节点才能稳定存在，整个集群才能稳定运行。当 Leader 节点崩溃之后，整个集群会有大约相当于选举超时的时间不可用，这种情况占比整个集群稳定运行的时间还是非常小的。

广播时间和平均故障间隔时间是由网络和服务器本身决定的，但是选举超时时间是可以由我们自己调节的。一般情况下，广播时间可以做到 0.5ms～50ms，选举超时时间设置为 200ms～1s 之间，而大多数服务器的平均故障间隔时间都在几个月甚至更长，很容易满足上述不等式的时间需求。

2.2　日志复制

通过上一节介绍的 Leader 选举过程，集群中最终会选举出一个 Leader 节点，而集群中剩余的其他节点将会成为 Follower 节点。Leader 节点除了向 Follower 节点发送心跳消息，还会处理客户端的请求，并将客户端的更新操作以消息（Append Entries 消息）的形式发送到集群中所有的 Follower 节点。当 Follower 节点记录收到的这些消息之后，会向 Leader 节点返回相应的响应消息。当 Leader 节点在收到半数以上的 Follower 节点的响应消息之后，会对客户端的请求进行应答。最后，Leader 会提交客户端的更新操作，该过程会发送 Append Entries 消息到 Follower 节点，通知 Follower 节点该操作已经提交，同时 Leader 节点和 Follower 节点也就可以将该操作应用到自己的状态机中。

上面这段描述仅仅是 Raft 协议中日志复制部分的大致流程，下面我们依然通过一个示例描

述该过程，为了方便描述，我们依然假设当前集群中有三个节点（A、B、C），其中 A 是 Leader 节点，B、C 是 Follower 节点，此时有一个客户端发送了一个更新操作到集群，如图 2-11（1）所示。前面提到过，集群中只有 Leader 节点才能处理客户端的更新操作，这里假设客户端直接将请求发给了节点 A。当收到客户端的请求时，节点 A 会将该更新操作记录到本地的 Log 中，如图 2-11（2）所示。

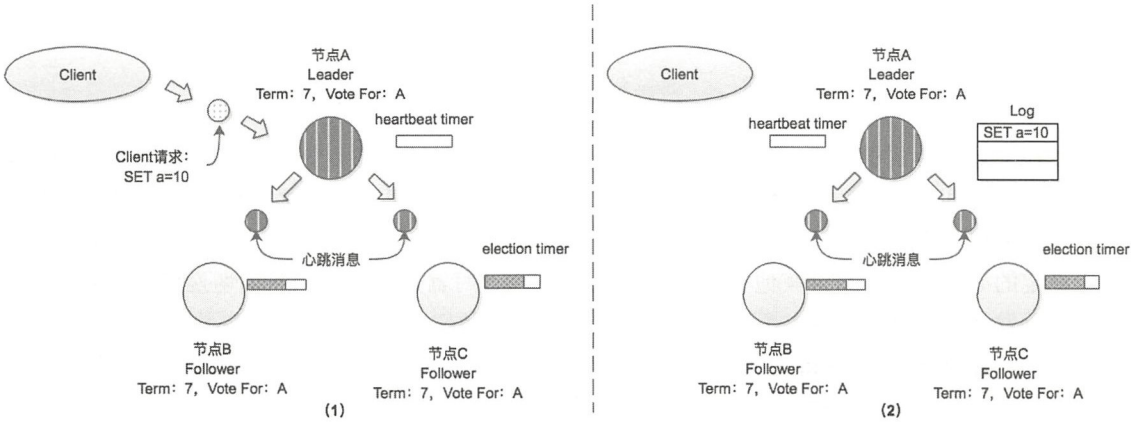


图 2-11

之后，节点 A 会向其他节点发送 Append Entries 消息，其中记录了 Leader 节点最近接收到的请求日志，如图 2-12（1）所示。集群中其他 Follower 节点收到该 Append Entries 消息之后，会将该操作记录到本地的 Log 中，并返回相应的响应消息，如图 2-12（2）所示。

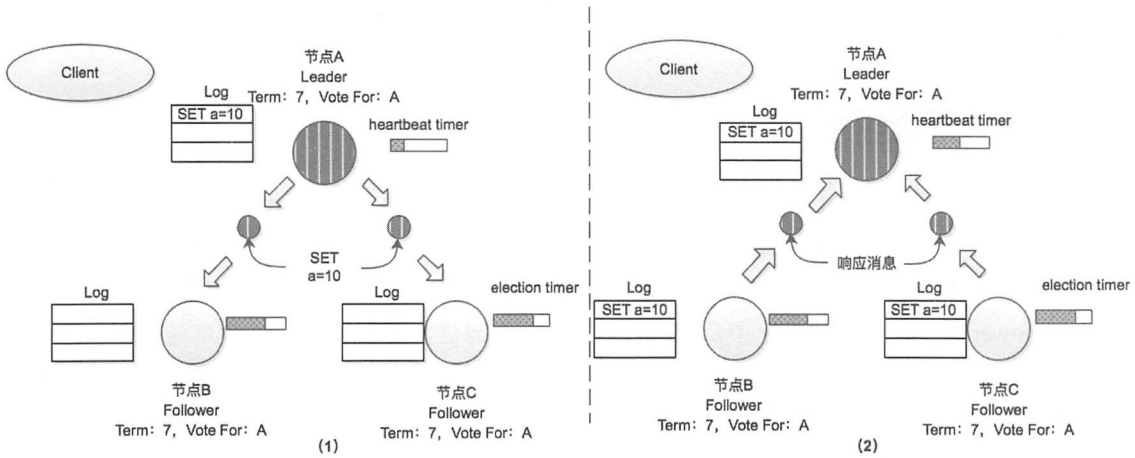


图 2-12

当 Leader 节点收到半数以上的响应消息之后，会认为集群中有半数以上的节点已经记录了

仅供非商业用途或交流学习使用

该更新操作，Leader 节点会将该更新操作对应的日志记录设置为已提交（committed），并应用到自身的状态机中。同时 Leader 节点还会对客户端的请求做出响应，如图 2-13（1）所示。同时，Leader 节点也会向集群中的其他 Follower 节点发送消息，通知它们该更新操作已经被提交，Follower 节点收到该消息之后，才会将该更新操作应用到自己的状态机中，如图 2-13（2）所示。

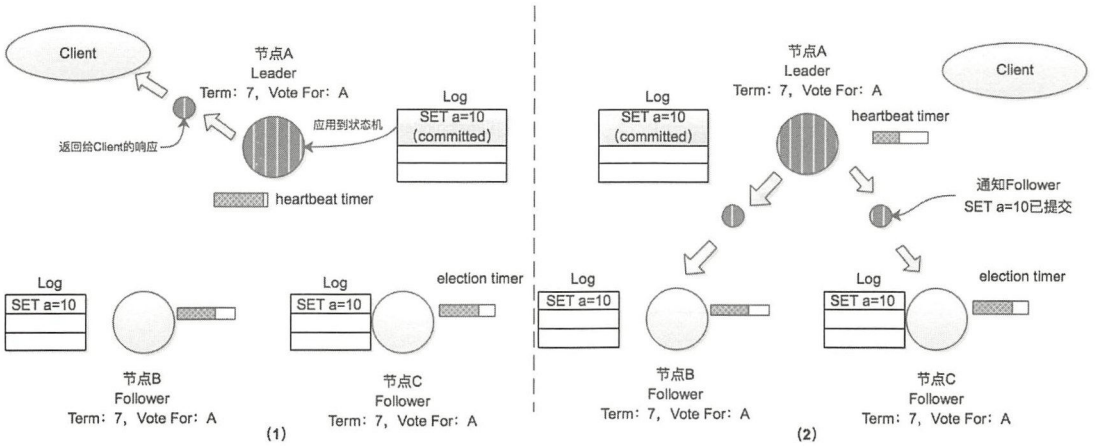


图 2-13

在上述示例的描述中我们可以看到，集群中各个节点都会维护一个本地 Log 用于记录更新操作，除此之外，每个节点还会维护 commitIndex 和 lastApplied 两个值，它们是本地 Log 的索引值，其中 commitIndex 表示的是当前节点已知的、最大的、已提交的日志索引值，lastApplied 表示的是当前节点最后一条被应用到状态机中的日志索引值。当节点中的 commitIndex 值大于 lastApplied 值时，会将 lastApplied 加 1，并将 lastApplied 对应的日志应用到其状态机中。

在 Leader 节点中不仅需要知道自己的上述信息，还需要了解集群中其他 Follower 节点的这些信息，例如，Leader 节点需要了解每个 Follower 节点的日志复制到哪个位置，从而决定下次发送 Append Entries 消息中包含哪些日志记录。为此，Leader 节点会维护 nextIndex[]和 matchIndex[]两个数组，这两个数组中记录的都是日志索引值，其中 nextIndex[]数组记录了需要发送给每个 Follower 节点的下一条日志的索引值，matchIndex[]表示记录了已经复制给每个 Follower 节点的最大的日志索引值。

这里简单看一下 Leader 节点与某一个 Follower 节点复制日志时，对应 nextIndex 和 matchIndex 值的变化：Follower 节点中最后一条日志的索引值大于等于该 Follower 节点对应的 nextIndex 值，那么通过 Append Entries 消息发送从 nextIndex 开始的所有日志。之后，Leader 节点会检测该 Follower 节点返回的相应响应，如果成功则更新相应该 Follower 节点对应的 nextIndex 值和 matchIndex 值；如果因为日志不一致而失败，则减少 nextIndex 值重试。

仅供非商业用途或交流学习使用

下面我们依然通过一个示例来说明 nextIndex[]和 matchIndex[]在日志复制过程中的作用，假设集群现在有三个节点，其中节点 A 是 Leader 节点（Term=1），而 Follower 节点 C 因为宕机导致有一段时间未与 Leader 节点同步日志。此时，节点 C 的 Log 中并不包含全部的已提交日志，而只是节点 A 的 Log 的子集，节点 C 故障排除后重新启动，当前集群的状态如图 2-14 所示（这里只关心 Log、nextIndex[]、matchIndex[]，其他的细节省略，另外需要注意的是，图中的 Term=1 表示的是日志发送时的任期号，而非当前的任期号）。

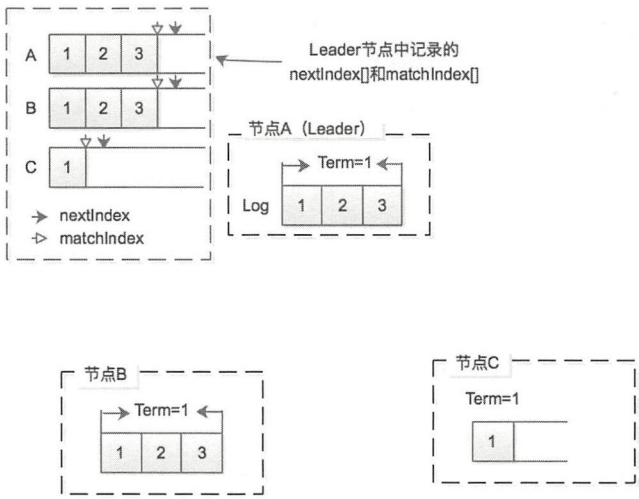


图 2-14

A 作为 Leader 节点，记录了 nextIndex[]和 matchIndex[]，所以知道应该向节点 C 发送哪些日志，在本例中，Leader 节点在下次发送 Append Entries 消息时会携带 Index=2 的消息（这里为了描述简单，每条消息只携带单条日志，Raft 协议采用批量发送的方式，这样效率更高），如图 2-15（1）所示。当节点 C 收到 Append Entries 消息后，会将日志记录到本地 Log 中，然后向 Leader 节点返回追加日志成功的响应，当 Leader 节点收到响应之后，会递增节点 C 对应的 nextIndex 和 matchIndex，这样 Leader 节点就知道下次发送日志的位置了，该过程如图 2-15（2）所示。

在上例中，当 Leader 节点并未发生过切换，所以 Leader 节点始终准确地知道节点 C 对应 nextIndex 值和 matchIndex 值。

如果在上述示例中，在节点 C 故障恢复后，节点 A 宕机后重启，并且导致节点 B 成为新任期（Term=2）的 Leader 节点，则此时节点 B 并不知道旧 Leader 节点中记录的 nextIndex[]和 matchIndex[]信息，所以新 Leader 节点会重置 nextIndex[]和 matchIndex[]，其中会将 nextIndex[]全部重置为其自身 Log 的最后一条已提交日志的 Index 值，而 matchIndex[]全部重置为 0，如图 2-16 所示。

仅供非商业用途或交流学习使用

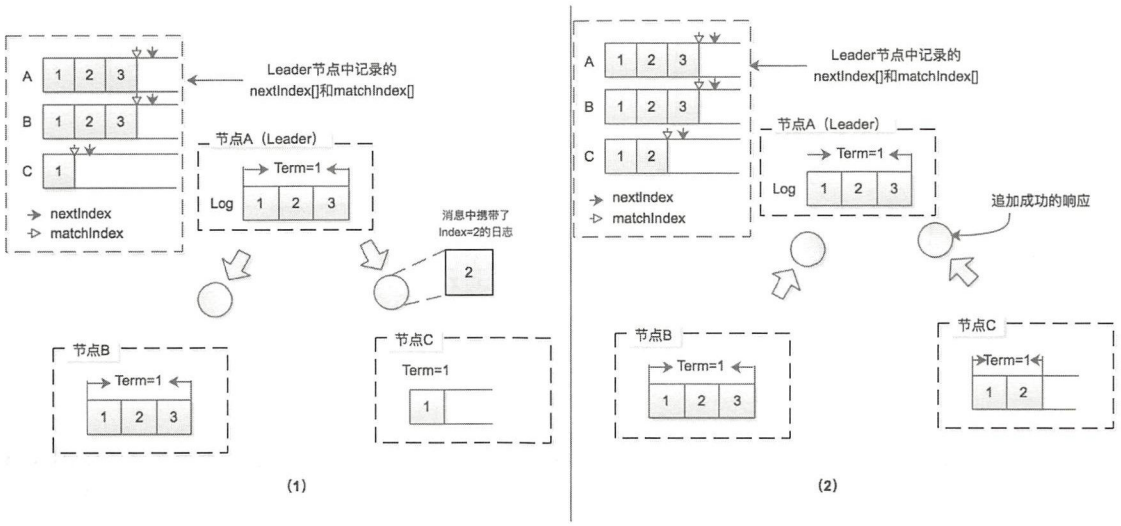


图 2-15

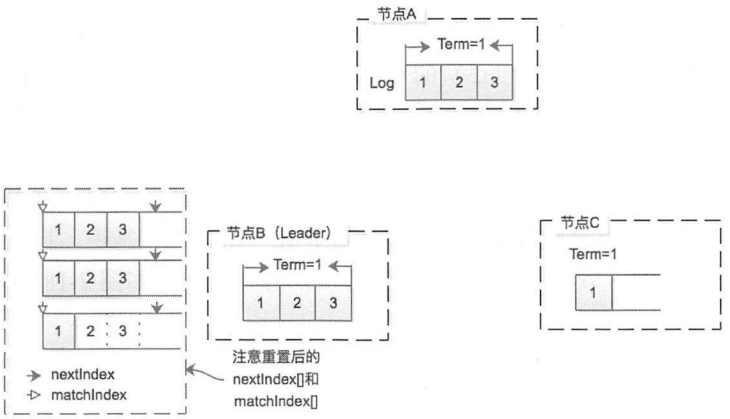


图 2-16

随后，新任期中的 Leader 节点会向其他节点发送 Append Entries 消息，如图 2-17（1）所示，节点 A 已经拥有了当前 Leader 的全部日志记录，所以会返回追加成功的响应并等待后续的日志，而节点 C 并没有 Index=2 和 Index=3 两条日志，所以返回追加日志失败的响应，在收到该响应后，Leader 节点会将 nextIndex 前移，如图 2-17（2）所示。

仅供非商业用途或交流学习使用

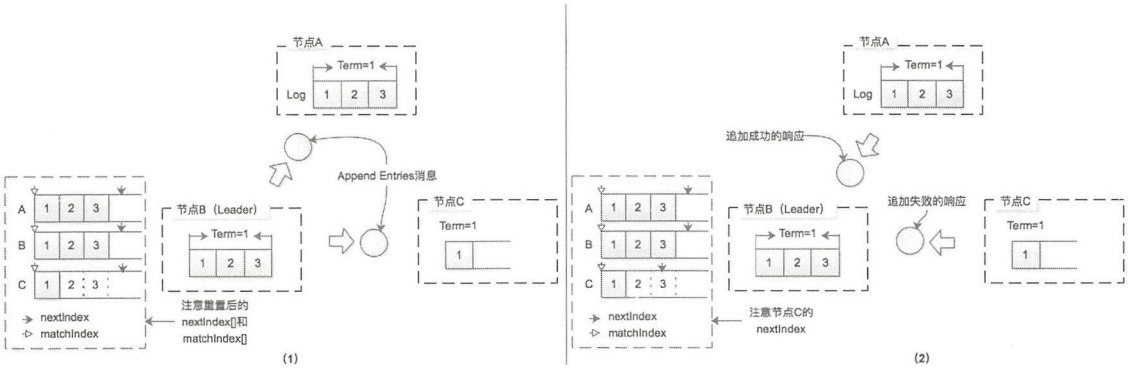


图 2-17

然后新 Leader 节点会再次尝试发送 Append Entries 消息，循环往复，不断减小 nextIndex 值，直至节点 C 返回追加成功的响应，之后就进入了正常追加消息记录的流程，不再赘述。

了解了 Log 日志及节点中基本的数据结构之后，请读者回顾前面描述的选举过程，其中 Follower 节点的投票过程并不像前面描述的那样简单（先收到哪个 Candidate 节点的选举请求，就将选票投给哪个 Candidate 节点），Follower 节点还需要比较该 Candidate 节点的日志记录与自身的日志记录，拒绝那些日志没有自己新的 Candidate 节点发来的投票请求，确保将选票投给包含了全部已提交（committed）日志记录的 Candidate 节点。这也就保证了已提交的日志记录不会丢失：Candidate 节点为了成为 Leader 节点，必然会在选举过程中向集群中半数以上的节点发送选举请求，因为已提交的日志记录必须存在集群中半数以上的节点中，这也就意味着每一条已提交的日志记录肯定在这些接收到节点中的至少存在一份。也就是说，记录全部已提交日志的节点和接收到 Candidate 节点的选举请求的节点必然存在交集，如图 2-18 所示。

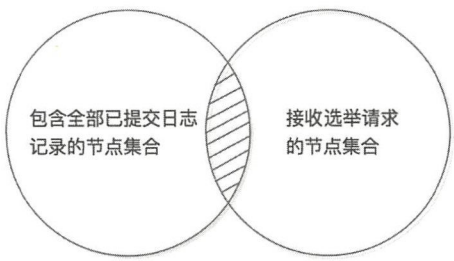


图 2-18

如果 Candidate 节点上的日志记录与集群中大多数节点上的日志记录一样新，那么其日志一定包含所有已经提交的日志记录，也就可以获得这些节点的投票并成为 Leader。

在比较两个节点的日志新旧时，Raft 协议通过比较两节点日志中的最后一条日志记录的索引值和任期号，以决定谁的日志比较新：首先会比较最后一条日志记录的任期号，如果最后的

仅供非商业用途或交流学习使用

日志记录的任期号不同，那么任期号大的日志记录比较新；如果最后一条日志记录的任期号相同，那么日志索引较大的比价新。

这里只是大概介绍一下 Raft 协议的流程和节点使用的各种数据结构，读者需要了解的是 Raft 协议的工作原理，如果对上述数据结构描述感到困惑，在后面介绍 etcd-raft 模块时，还会再次涉及这些数据结构，到时候读者可以结合代码及这里的描述进一步进行分析。

2.3　网络分区的场景

接下来，我们来看一下 Raft 协议是如何处理网络分区情况的。在一个集群中，如果有部分节点的网络发生故障，与集群中另一部分节点的连接中断，就会出现网络分区，如图 2-19 所示，集群有 A、B、C、D、E 五个节点，其中节点 A、B 相互之间网络连通，节点 C、D、E 相互之间网络连通，但是这两部分节点之间出现网络故障，这就形成了网络分区。

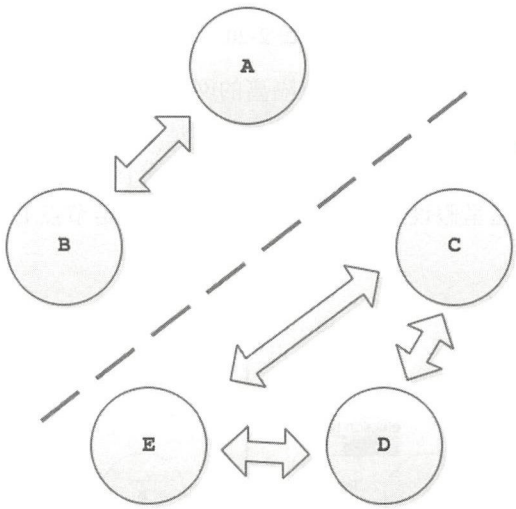


图 2-19

这里依然通过一个示例来说明 Raft 协议对网络分区场景的处理。假设集群中节点 A 是 Leader 节点，它会向其他四个节点发送 Append Entries 消息和心跳消息，如图 2-20（1）所示。当出现网络出现分区时，节点 A 的心跳消息只有节点 B 才能收到，而集群中的其他节点收不到，如图 2-20（2）所示（图中节点 A 发往节点 C、D、E 的消息由于网络分区，并不会抵达节点 C、D、E，故未在图中画出）。

仅供非商业用途或交流学习使用

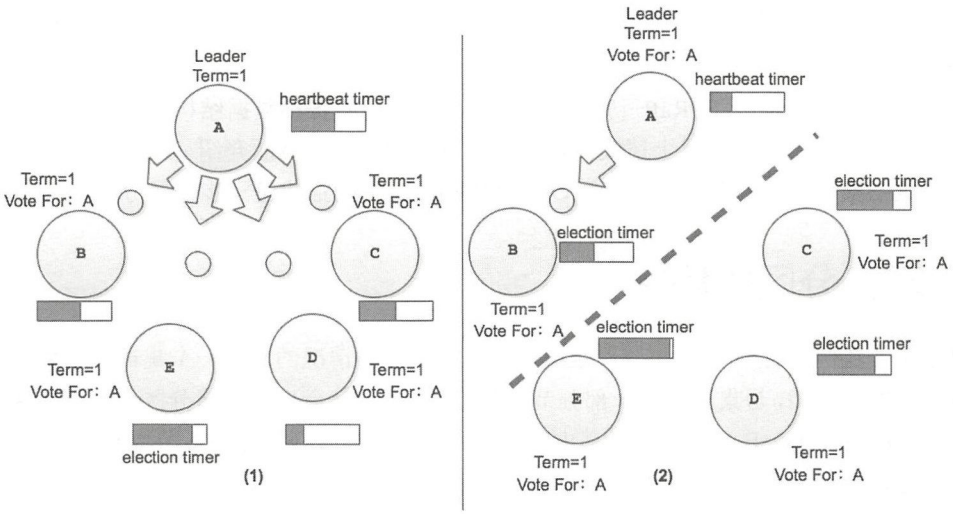


图 2-20

随着时间的流逝，集群中与 Leader 节点隔离的网络分区（C、D、E）中，会率先有一个节点的选举计时器（election timer）超时，这里假设该节点是 E，此时的节点 E 就会切换成 Candidate 状态并发起下一轮选举，如图 2-21（1）所示。由于网络分区，当前集群中只有节点 C、D 能够收到节点 E 的选举请求，这里假设节点 C、D 都会将选票投给节点 E，如图 2-21（2）所示。

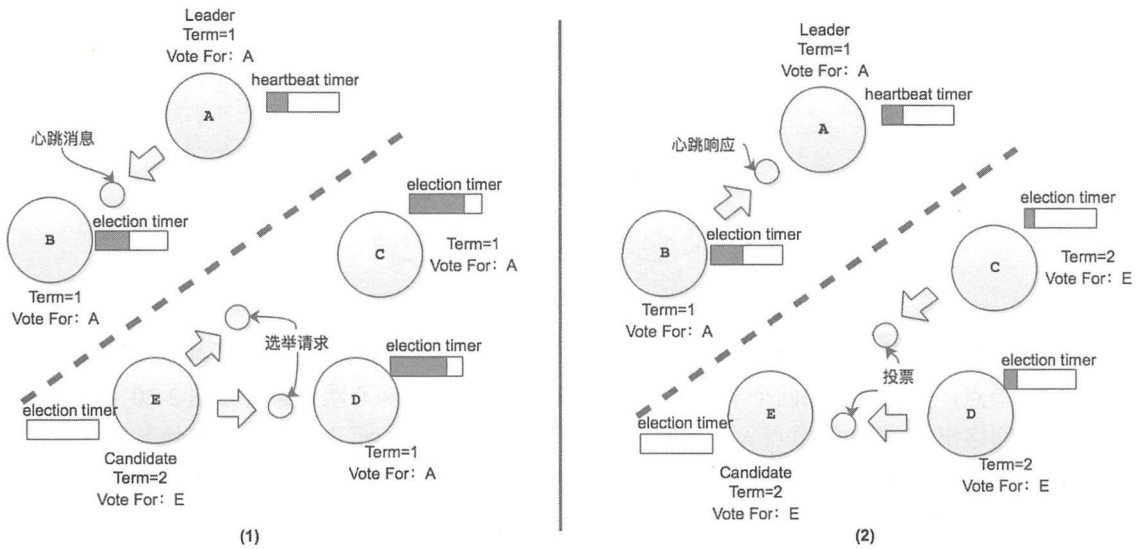


图 2-21

到此为止，节点 E 在此次选举中收到了得到三票（其中包括它本身的一票），达到集群半

仅供非商业用途或交流学习使用

数以上，所以节点 E 成为新任期（Term=2）的 Leader 节点，如图 2-22 所示：

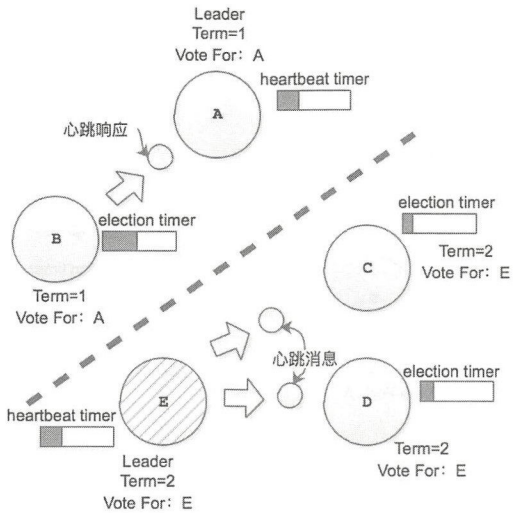


图 2-22

当网络故障被修复时，上述的网络分区也就会消失，此时节点 A（任期 Term=1 的 Leader 节点）发送的心跳消息会被节点 C、D、E 接收到（图 2-22 中虽然省略了这些由于网络分区而无法送达的心跳消息，但实际上节点 A 依然认为自己是 Leader 节点，在发送心跳消息时也会向节点 C、D、E 发送心跳消息），但是这些心跳消息中携带的 Term 值小于当前 C、D、E 节点的 Term 值，会被 C、D、E 节点忽略；同时，节点 E（Term=2 任期的 Leader 节点）发送的心跳消息会被节点 A、B 接收到（图 2-22 中同样省略了这些无法送达的心跳消息），不同的是，这些心跳消息携带的 Term 值大于当前 A、B 节点的 Term 值，所以节点 A、B 会切换成 Follower 状态，这样整个集群中的 Leader 节点依然是节点 E。

读者可能会问：如果网络分区时，Leader 节点划分到节点较多的分区中，如图 2-23 所示，此时节点较少的分区中，会有节点的选举计时器超时，切换成 Candidate 状态并发起新一轮的选举。但是由于该分区中节点数不足半数，所以无法选举出新的 Leader 节点。待一段时间之后，该分区中又会出现某个节点的选举计时器超时，会再次发起新一轮的选举，循环往复，从而导致不断发起选举，Term 号不断增长。

在 Raft 协议中对这种情况有一个优化，当某个节点要发起选举之前，需要先进入一个叫作 PreVote 的状态，在该状态下，节点会先尝试连接集群中的其他节点，如果能够成功连接到半数以上的节点，才能真正发起新一轮的选举。通过这种方式就可以解决上述的问题，在后面分析 etcd-raft 模块时，还会详细介绍其具体实现。

仅供非商业用途或交流学习使用

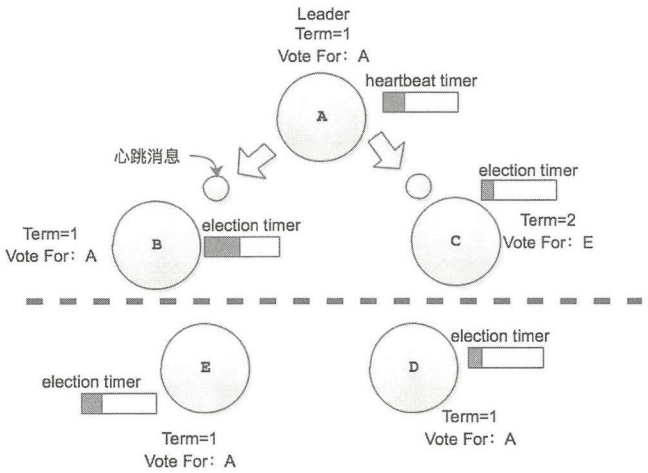


图 2-23

回到前面的示例简单来介绍网络分区恢复时的相关处理。当网络分区恢复时，集群中存在新旧两个 Leader 节点（A 和 E），其中节点 E 的 Term 值较高，会成为整个集群中的 Leader 节点。但是由于之前的网络分区，节点 A、B 的本地 Log 中可能存在未提交的日志记录，如图 2-24（1）所示，此时节点 A 和 B 会回滚未提交的日志记录，并重新复制新 Leader 节点的日志，如图 2-24（2）所示。

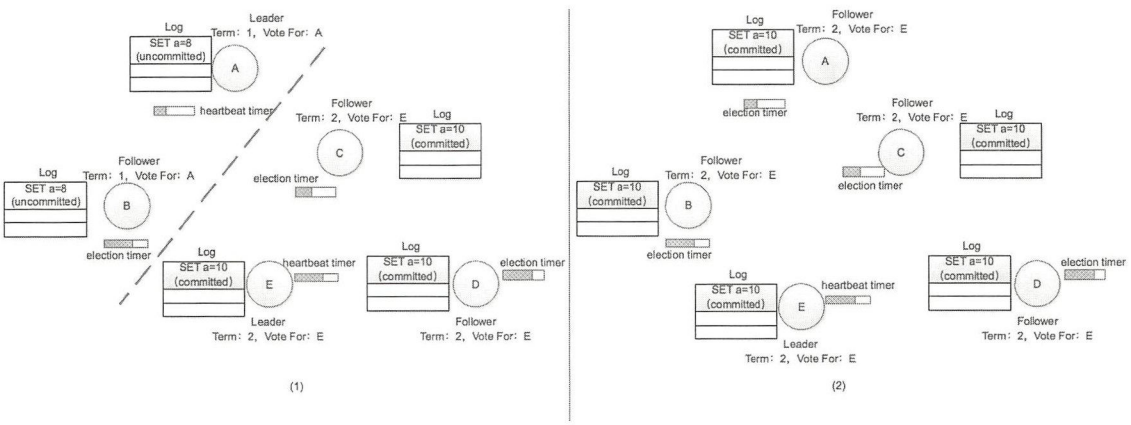


图 2-24

这样在网络分区恢复之后，整个集群的日志又会恢复一致。到此为止，网络分区场景下的 Leader 选举及日志复制过程就介绍完了，希望通过对这种特殊场景的介绍，读者能够更深刻地了解 Raft 协议的工作原理。

另一个需要介绍的问题是，网络分区场景下，客户端与集群的交互过程及日志复制的过程。

仅供非商业用途或交流学习使用

这里我们先简单介绍一下客户端如何与集群进行交互并找到集群的 Leader 节点。在前面提到过，集群中只有 Leader 节点可以处理客户端发来的请求，当 Follower 节点收到客户端的请求时，也必须将 Leader 节点信息告知客户端，然后由 Leader 节点处理其请求，具体步骤如下：

（1）当客户端初次连接到集群时，会随机挑选一个服务器节点进行通信。

（2）如果客户端第一次挑选的节点不是 Leader 节点，那么该节点会拒绝客户端的请求，并且将它所知道的 Leader 节点的信息返回给客户端。

（3）当客户端连接到 Leader 节点之后，即可发送消息进行交互。

（4）如果在交互过程中 Leader 节点宕机，那么客户端的请求会超时，客户端会再次随机挑选集群中的节点，并从步骤 1 重新开始执行。

这里依然通过一个示例来介绍整个过程，假设集群依然有五个节点，在未发生网络分区时，节点 A 为集群的 Leader 节点，此时的客户端请求会发送到节点 A，经过前面描述的日志复制过程后，节点 A 也会向客户端返回响应，与如图 2-25（1）和（2）所示。

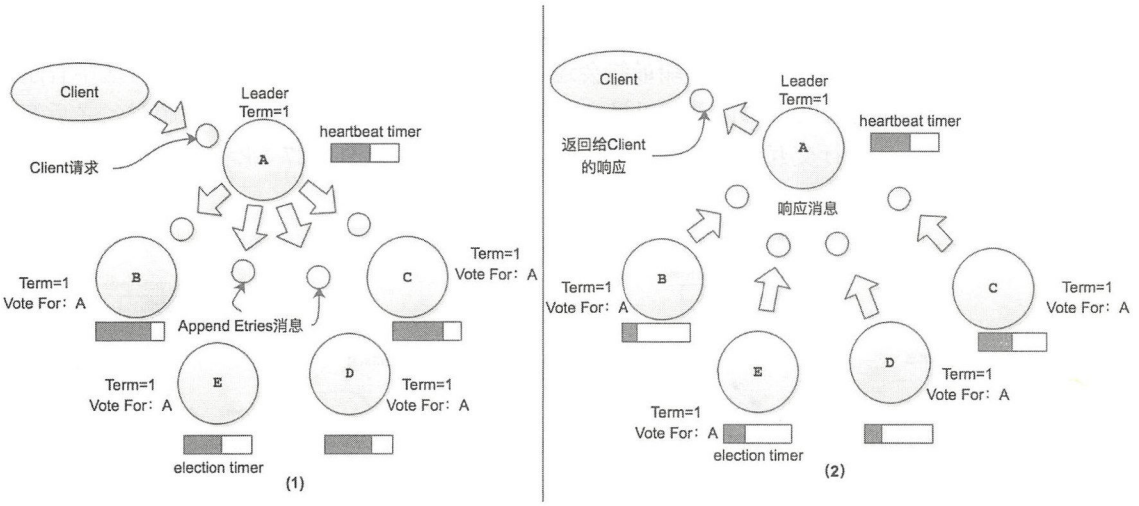


图 2-25

当节点 A、B 与节点 C、D、E 之间发生网络分区之后，客户端发往节点 A 的请求将会超时，这主要是因为节点 A 无法将请求发送到集群中超过半数的节点上，该请求相应的日志记录也就无法提交，从而导致无法给客户端返回相应的响应，该过程如图 2-26（1）和（2）所示。

仅供非商业用途或交流学习使用

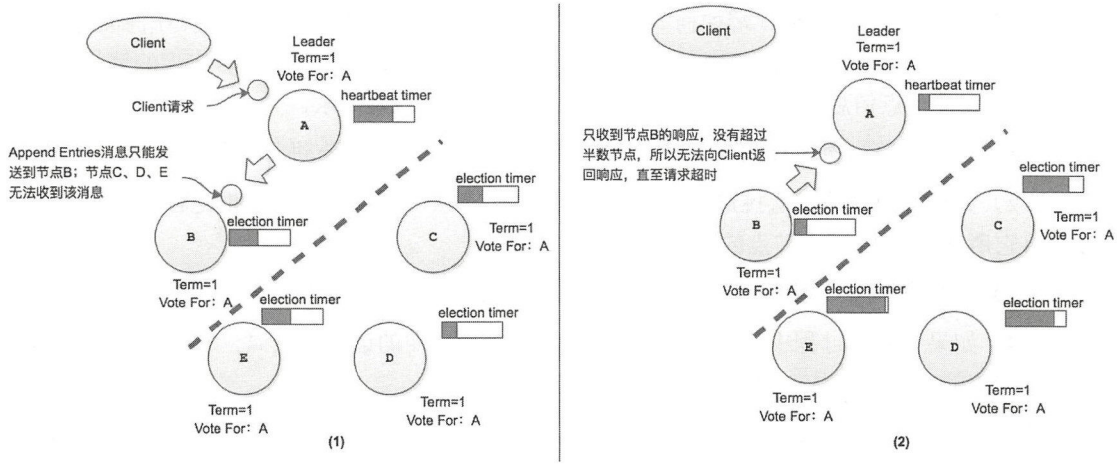


图 2-26

前面已经介绍了网络分区之后的 Leader 选举过程，这里不再赘述，该示例中假设节点 E 被选举为新任期（Term=2）的 Leader 节点。当请求超时之后，客户端会重新随机选择一个节点，并获取新 Leader 节点的信息，客户端最终会连接到节点 E 并发送请求，而该网络分区中有超过半数的节点，请求对应的日志记录可以提交，所以客户端的请求不会再次出现超时，之后客户端会一直与节点 E 进行交互直至下次请求超时。上述过程的如图 2-27（1）～（4）所示。

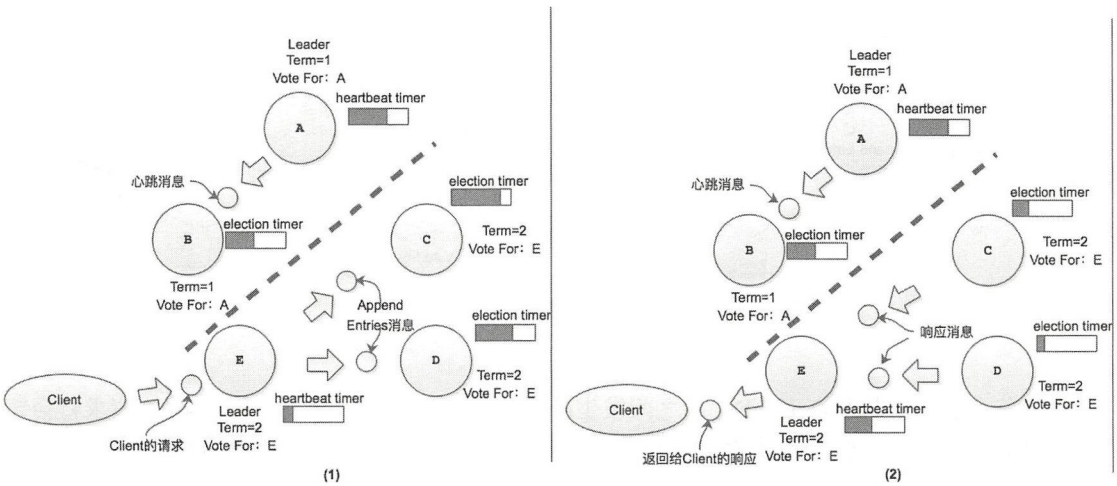


图 2-27

仅供非商业用途或交流学习使用

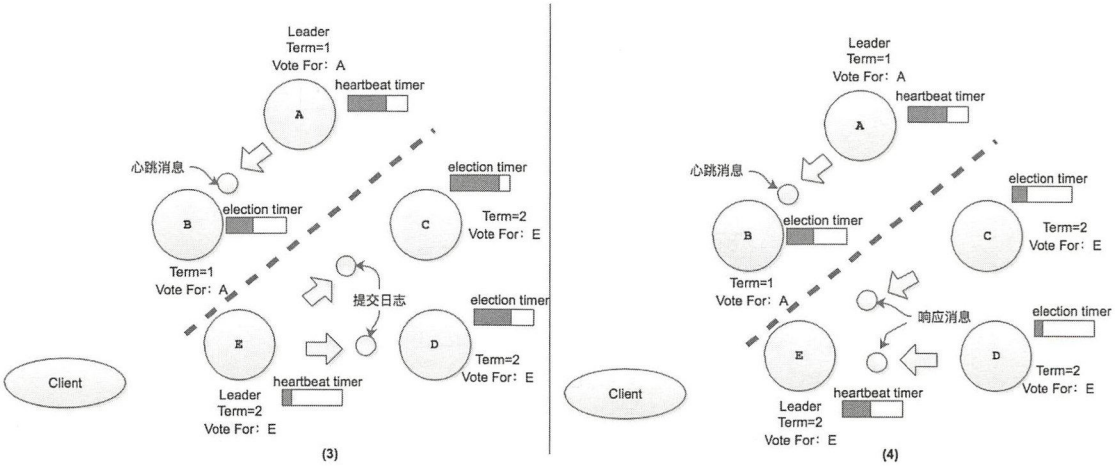


图 2-27（续）

在 Raft 协议的论文中，还给出了另一种 proxy 的方案：假设客户端连接到集群中的某个 Follower 节点，该 Follower 节点会将客户端发送的所有请求转发给 Leader 节点进行处理，当 Leader 节点响应 Follower 节点之后，再由 Follower 节点响应客户端。当出现请求超时的情况时，客户端同样需要随机选择新的节点进行连接。

2.4　日志压缩与快照

通过前面章节的描述可知，随着客户端与集群不断地交互，每个节点上的日志记录会不断增加，但是服务器的空间都是有限的，日志量不能无限制地增长。另外，在节点重启时会重放日志记录，如果日志记录过多，则需要花费较长的时间完成重放操作。这就需要压缩和清除机制来减少日志量，从而避免上述情况。

定期生成快照是最常见也是最简单的压缩方法。在创建快照文件时，会将整个节点的状态进行序列化，然后写入稳定的持久化存储中，这样，在该快照文件之前的日志记录就可以全部丢弃了。例如，集群中变量 a 的值为 100，客户端发送了一个更新请求将变量 a 更新为 13，经过前面描述的日志复制过程之后，该请求对应的日志记录最终被提交并应用到集群中的每个节点中。此时每个节点中维护的变量 a 都是 13，而 a=13 这条日志记录就无须继续保留。在 ZooKeeper、Chubby 和 etcd 中都有类似上述的快照处理逻辑，这里只是介绍创建快照文件和压缩日志的基本逻辑，在后面的章节会具体介绍其实现。

在快照中除了节点当前的数据状态，还包含了其最后一条日志记录的任期号和索引号，如图 2-28 所示，该快照包含了 6 条日志记录，在快照的元数据中记录了第 6 条日志记录的任期号和索引号，在生成快照文件之后，即可将 1～6 条日志记录丢弃了。

仅供非商业用途或交流学习使用

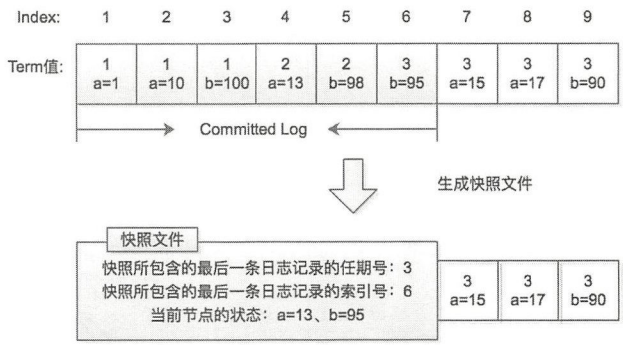


图 2-28

一般情况下，集群中的每个节点都会自己独立、定时地创建快照，在其状态恢复时，都会使用自己本地最新的快照数据。如果 Follower 节点长时间宕机（或是刚刚加入集群的新节点），就有可能导致其日志记录远远落后于当前的 Leader 节点，与此同时，Leader 节点中陈旧的日志记录已被删除了。在这种场景下，为了将该 Follower 节点恢复到正确的状态，Leader 节点会将快照发送给该 Follower 节点，Follower 节点会使用该快照数据进行状态恢复。当 Leader 节点需要向 Follower 节点发送快照时，会发送一种特殊的消息类型（快照消息）。etcd 的网络层为了高效地传输消息，会将快照的发送与普通消息（Append Entries 消息、心跳消息等）的发送分开在不同的消息通道中完成，在后面介绍 etcd 网络层时会详细介绍。

当 Follower 节点接收到该快照消息时，必须决定如何处理已存在的日志记录，在快照中之所以保留前面介绍的一些元数据，其作用之一就是为了在 Follower 节点收到快照之后进行一致性检查。一般情况下，快照已包含了该 Follower 节点中不存在的日志记录，此时 Follower 节点直接丢弃其所有的日志记录，因为这些日志最终会被 Leader 传递来的快照所代替。如果 Follower 节点接收到的快照只包含了自己本地日志的一部分，那么被该快照所包含的全部日志记录会被全部删除，但是快照之后的日志则会保留。

有的读者可能会考虑过另一种替代方案，即只有 Leader 节点创建快照，然后发送给所有的 Follower 节点。但是该方案有几个缺点：首先就是快照数据会比较大，并且发送快照数据是比较浪费网络带宽的，也比较耗时，这显然比 Follower 节点从本地直接加载要耗时很多；其次就是 Leader 的实现会更加复杂。

在 Raft 协议中，每个节点都会创建快照，所以创建快照的时机决定了快照的性能。如果创建快照过于频繁，那么就会消耗大量的资源，导致每个节点的性能下降；如果创建快照的频率过低，那么两次创建快照之间积累的日志记录会比较多，快照就无法为节点节约内存等资源。所以我们要在两者之间进行权衡，常见的策略是当日志记录个数达到一个固定阈值的时候，就触发一次创建快照的操作，生成相应的快照文件，我们可以通过调节该阈值来控制创建快照的频率。

仅供非商业用途或交流学习使用

2.5 其他技术点

2.5.1 linearizable语义

Raft 协议的目标是实现 linearizable 语义，即在客户端每次向集群发送一次读请求时，该请求只会被执行一次。但是根据前面的描述，客户端虽然只是想发送一次请求，但是集群可能多次收到该请求。例如，Leader 节点负责提交日志记录（通知了其他 Follower 节点）并将日志记录应用到了其状态机中，但是在向客户端返回相应的响应消息之前宕机了，那么客户端会连接到新的 Leader 节点并重发对应的请求，这就导致该请求再次被执行。或者，网络出现故障，导致请求丢失或是延迟，如图 2-29 所示，就会导致同一条请求被执行两次。

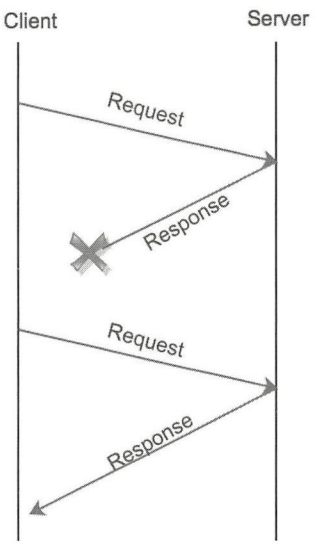


图 2-29

常见的解决方案就是客户端对于每个请求都产生一个唯一的序列号，然后由服务端为每个客户端维护一个 Session，并对每个请求进行去重。当服务端接收到一个请求时，会检测其序列号，如果该请求已经被执行过了，那么就立即返回结果，而不会重新执行。

2.5.2 只读请求

在介绍 Raft 协议的日志复制时提到，请求对应的日志记录会写入 Leader 节点的本地 Log 中并完成复制到集群中半数以上的节点，之后才会真正提交并应用到状态机中。为了提高只读

仅供非商业用途或交流学习使用

请求的性能，我们可以考虑直接处理而不记录对应的日志记录（也不会经过日志复制的过程）。但是，在不增加任何限制的情况下，这么做可能会冒着返回脏数据的风险，因为 Leader 节点响应客户端请求时可能已经故障（或是已经发生了网络分区），集群已经选出了新的 Leader 节点，但是旧的 Leader 节点自身还不知道。

为了不返回脏数据，同时为了保证 linearizability 语义，Raft 协议在处理只读请求时，除了直接读取 Leader 节点对应的状态信息，还需要使用额外的措施。处理只读请求的大致逻辑如下：

（1）Leader 节点必须有关于已提交日志的最新信息，虽然在节点刚刚赢得选举成为 Leader 时，拥有所有已经被提交的日志记录，但是在其任期刚开始时，它可能不知道哪些是已经被提交的日志。为了明确这些信息，它会在其任期开始时提交一条空日志记录，这样上一个任期中的所有日志都会被提交。

（2）Leader 节点会记录该只读请求对应的编号作为 readIndex，当 Leader 节点的提交位置（commitIndex）达到或是超过该位置之后，即可响应该只读请求。

（3）Leader 节点在处理只读的请求之前必须检查集群中是否有新的 Leader 节点，自己是否已经被作废，如果该节点已经不再是集群的 Leader 节点，则该节点中日志记录就可能包含脏数据，必须由新 Leader 节点来处理此次只读请求。Raft 协议中，通过让 Leader 节点在处理只读请求之前，先和集群中的半数以上的节点交换一次心跳消息来解决这个问题。如果该 Leader 节点可以与集群中半数以上的节点交换一次心跳，则表示该 Leader 节点依然为该集群最新的 Leader 节点。这样，readIndex 值也就是整个集群中所有节点所能看到的最大提交位置（commitIndex）。

（4）随着日志记录的不断提交，Leader 节点的提交位置（commitIndex）最终会超过上述 readIndex，此时 Leader 就可以响应客户端的只读请求了。

这里简单介绍一下 linearizability 的含义，线性化（linearizability）是分布式系统中比较重要的概念。linearizability 是对单对象上的单个操作的一种顺序保证，它提供了对于同一个对象的一系列读写操作都是按照实时时间排序的保证。简单地说，linearizability 保证对于一个对象的写操作一旦完成，需要立即被后续的读操作看到，即读操作一定是读到该对象的最新的值。从该角度来看，linearizability 与 atomic consistency 是同义词，也是 CAP 原则中的 C（consistency）。另外，并且 linearizability 是可组合的，如果系统中每个对象的操作都是 linearizable，则系统中所有操作是 linearizable。

2.5.3 PreVote状态

在前面介绍网络分区场景时提到，在节点数不足集群半数的网络分区中，始终没有节点可

仅供非商业用途或交流学习使用

以获取半数以上的选票成为 Leader 节点，所以每过一段时间，就有节点的选举计时器超时并切换成 Candidate 状态，发起新一轮的选举。

这虽然不影响集群的使用（在节点超过半数的网络分区中，已经成功选举出 Leader 节点并对外提供服务），但是会导致不断发起选举的节点的 Term 号不断增长。当网络分区结束时，由于该节点的 Term 值高于集群当前的 Leader 节点的 Term 值，就会迫使当前 Leader 节点发生状态切换，并重新发起一次新的选举。

Raft 协议为了优化此次无意义的选举，给节点添加了一个 PreVote 的状态：当某个节点要发起选举之前，需要先进入 PreVote 的状态。在 PreVote 状态下的节点会先尝试连接集群中的其他节点，如果能够成功连接到半数以上的节点，才能真正切换成 Candidate 状态并发起新一轮的选举。在后面分析 etcd-raft 模块时，我们可以看到相关实现。

2.5.4　Leader节点转移

通过前面的介绍我们知道，Leader 节点在整个集群中的作用至关重要。但是在有的场景中需要对 Leader 节点进行手动切换。例如，我们要将 Leader 节点所在的机器进行系统升级或是停机维护等。此时，我们可能需要集群中指定的 Follower 节点成为新的 Leader 节点，继续对外提供服务。在原 Leader 节点所在的机器维护结束之后，我们可能还需要将 Leader 节点再转移到该机器上（可能该机器的配置等条件优于集群中的其他机器，更适合做 Leader 节点）。这种场景下就需要特定的 Follower 节点成为下一任期的 Leader 节点。根据前面介绍的 Leader 选举过程我们知道，Leader 节点的选举在本质上是随机的，无法满足上述需求。

Raft 协议给出的方案是：首先暂停接收客户端请求，让一个指定的 Follower 节点的本地日志与当前的 Leader 节点完全同步，在完成同步之后，该特定的 Follower 节点立刻发起新一轮的选举。由于其 Term 值较大，原 Leader 节点自然被其替换下来。该方案需要控制好选举计时器及特定 Follower 与 Leader 节点同步的时间，防止其他 Follower 节点在这段时间内发起选举，当然，发生这种情况的概率还是比较低的。

在实现 Raft 协议的时候，除了上面提到的扩展点和优化点，在 Raft 大论文中还提到一些其他的相关内容，非常值得参考。笔者也极力推荐读者亲自阅读该论文，毕竟本书篇幅有限，无法将其内容逐一介绍。

本章小结

本章主要介绍了 Raft 协议的基本概念和基本流程，其中包括 Leader 节点的选举、节点间的

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

复制、日志的压缩、快照的生成，以及网络分区等场景的介绍，最后还介绍了在实现 Raft 协议时可能会遇到的特殊问题的相关方案。在本章中介绍每个流程时，都是通过示例方式进行描述的，希望读者在阅读完本章后，对 Raft 协议有一个初步的了解，为后面分析 etcd-raft 模块打下基础。在后面分析 etcd-raft 模块时，也建议读者回顾本章 Raft 算法的实例，了解 etcd 实现与 Raft 算法的细微差异。

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用



第 3 章
etcd-raft 模块详解

通过上一章对 Raft 协议的介绍，相信读者已经对 Raft 协议的关键流程和工作原理有了大致的了解。在本章中，我们将详细分析 etcd-raft 模块的实现，希望读者在分析 etcd-raft 模块的代码时，结合上一章对 Raft 协议的介绍，更加深入地了解 Raft 协议。

在开始介绍 etcd-raft 模块的具体实现之前，我们需要先来了解一下其中的一些概念，具体如下。

- Node：对 etcd-raft 模块具体实现的一层封装，方便上层模块使用 etcd-raft 模块。
- **上层模块**：etcd-raft 模块的调用者，上层模块通过 Node 提供的 API 与底层的 etcd-raft 模块进行交互。
- Cluster：表示一个集群，其中记录了该集群的基础信息。
- Member：组成 Cluster 的元素之一，其中封装了一个节点的基本信息。
- Peer：集群中某个节点对同一集群中另外一个节点的称呼，在后面还会介绍其具体实现。
- Entry 记录：在前面介绍 Raft 协议时提到，节点之间传递的是消息（Message），每条消息中可以携带多条 Entry 记录，每条 Entry 记录对应一个独立的操作。

  在 Entry 中其中封装了如下信息。

  - Term（uint64 类型）：该 Entry 所在的任期号。
  - Index（uint64 类型）：该 Entry 对应的索引号。
  - Type（EntryType 类型）：该 Entry 记录的类型。该字段有两个可选项：一个是 EntryNormal，表示普通的数据操作；另一个是 EntryConfChange，表示集群的变更操作。

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

- Data（[]byte 类型）：具体操作使用的数据。

另外，在每个节点中记录的本地 Log 的基本单位也是 Entry 记录。有的文章也会将 Entry 记录称为“日志记录”，在 etcd 中还有一个 WAL 日志的概念，这两者并非完全等价，所以请读者注意一下，避免两者混淆。

在 etcd-raft 模块中另一个比较重要结构体就是 raftpb.Message。在 etcd-raft 模块的实现中，Message 是所有消息的抽象，包括了各种类型消息所需要的字段，其中核心字段的含义如下。

- Type（MessageType 类型）：该字段定义了消息的类型，etcd-raft 的实现中就是通过该字段区分不同的消息并进行分类处理的，MessageType 中共定义了 19 种消息类型，后面会详细分析每种消息类型的含义及相应的处理方式。
- From（uint64 类型）：发送消息的节点 ID。在集群中，每个节点都拥有一个唯一 ID 作为标识。
- To（uint64 类型）：消息的目标节点 ID。
- Term（uint64 类型）：发送消息的节点的 Term 值。如果 Term 值为 0，则为本地消息，在 etcd-raft 模块的实现中，对本地消息进行特殊处理。例如，后面见到的 MsgHup 类型消息就是本地消息的一种。
- Entries（[]Entry 类型）：如果是 MsgApp 类型的消息，则该字段中保存了 Leader 节点复制到 Follower 节点的 Entry 记录。在其他类型消息中，该字段的含义后面会详细介绍。
- LogTerm（uint64 类型）：该消息携带的第一条 Entry 记录的 Term 值。
- Index（uint64 类型）：记录了一个索引值，该索引值的具体含义与消息的类型相关。例如，MsgApp 消息的 Index 字段保存了其携带的 Entry 记录（即 Entries 字段）中前一条记录的 Index 值，而 MsgAppResp 消息的 Index 字段则是 Follower 节点提示 Leader 节点下次从哪个位置开始发送 Entry 记录。
- Commit（uint64 类型）：消息发送节点的提交位置（commitIndex）。
- Snapshot（Snapshot 类型）：在传输快照时，该字段保存了快照数据。
- Reject（bool 类型）：主要用于响应类型的消息，表示是否拒绝收到的消息。例如，Follower 节点收到 Leader 节点发来的 MsgApp 消息，如果 Follower 节点发现 MsgApp 消息携带的 Entry 记录并不能直接追加到本地的 raftLog 中，则会将响应消息的 Reject 字段设置为 true，并且会在 RejectHint 字段中记录合适的 Entry 索引值，供 Leader 节点参考。
- RejectHint（uint64 类型）：在 Follower 节点拒绝 Leader 节点的消息之后，会在该字段记录一个 Entry 索引值供 Leader 节点。

仅供非商业用途或交流学习使用

- Context（[]byte 类型）：消息携带的一些上下文信息。例如，该消息是否与 Leader 节点转移相关。

3.1　raft结构体

在 etcd-raft 模块中，raft 结构体是其核心数据结构，在结构体 raft 中封装了当前节点所有的核心数据。本节会对 raft 结构体进行详细分析，首先来看其核心字段。

- id（uint64 类型）：当前节点在集群中的 ID。
- Term（uint64 类型）：当前任期号。如果 Message 的 Term 字段为 0，则表示该消息是本地消息，例如，后面提到的 MsgHup、MsgProp、MsgReadIndex 等消息，都属于本地消息。
- Vote（uint64 类型）：当前任期中当前节点将选票投给了哪个节点，未投票时，该字段为 None。
- raftLog(*raftLog 类型)：在前面介绍过，在 Raft 协议中的每个节点都会记录本地 Log，在 etcd-raft 模块中，使用结构体 raftLog 表示本地 Log，在 raftLog 中还涉及日志的缓存等相关内容，后面会展开详细介绍。
- maxInflight（int 类型）：对于当前节点来说，已经发送出去但未收到响应的消息个数上限。如果处于该状态的消息超过 maxInflight 这个阈值，则暂停当前节点的消息发送，这是为了防止集群中的某个节点不断发送消息，引起网络阻塞或是压垮其他节点，从而影响其他节点的正常运行。
- maxMsgSize（uint64 类型）：单条消息的最大字节数。
- prs（map[uint64]*Progress 类型）：在上一章介绍 Raft 协议时提到过，Leader 节点会记录集群中其他节点的日志复制情况（NextIndex 和 MatchIndex）。在 etcd-raft 模块中，每个 Follower 节点对应的 NextIndex 值和 MatchIndex 值都封装在 Progress 实例中，除此之外，每个 Progress 实例中还封装了对应 Follower 节点的相关信息，这里简单介绍一下其核心字段的含义。
  - Match（uint64 类型）：对应 Follower 节点当前已经成功复制的 Entry 记录的索引值。
  - Next（uint64 类型）：对应 Follower 节点下一个待复制的 Entry 记录的索引值。
  - State（ProgressStateType 类型）：对应 Follower 节点的复制状态，其可选项的含义后面详细介绍。
  - Paused(bool 类型)：当前 Leader 节点是否可以向该 Progress 实例对应的 Follower

仅供非商业用途或交流学习使用

节点发送消息。

- PendingSnapshot（uint64 类型）：当前正在发送的快照数据信息。
- RecentActive（bool 类型）：从当前 Leader 节点的角度来看，该 Progress 实例对应的 Follower 节点是否存活。
- ins（*inflights 类型）：记录了已经发送出去但未收到响应的消息信息。

- state（StateType 类型）：当前节点在集群中的角色，可选值分为 StateFollower、StateCandidate、StateLeader 和 StatePreCandidate 四种状态。
- votes（map[uint64]bool 类型）：在选举过程，如果当前节点收到了来自某个节点的投票，则会将 votes 中对应的值设置为 true，通过统计 votes 这个 map，就可以确定当前节点收到的投票是否超过半数。
- msgs（[]pb.Message 类型）：缓存了当前节点等待发送的消息。
- lead（uint64 类型）：当前集群中 Leader 节点的 ID。
- leadTransferee（uint64 类型）：用于集群中 Leader 节点的转移，leadTransferee 记录了此次 Leader 角色转移的目标节点的 ID。
- electionElapsed（int 类型）：选举计时器的指针，其单位是逻辑时钟的刻度，逻辑时钟每推进一次，该字段值就会增加 1。
- electionTimeout（int 类型）：选举超时时间，当 electionElapsed 字段值到达该值时，就会触发新一轮的选举。
- heartbeatElapsed（int 类型）：心跳计时器的指针，其单位也是逻辑时钟的刻度，逻辑时钟每推进一次，该字段值就会增加 1。
- heartbeatTimeout（int 类型）：心跳超时时间，当 heartbeatElapsed 字段值到达该值时，就会触发 Leader 节点发送一条心跳消息。
- checkQuorum（bool 类型）：在前面介绍 Raft 协议时提到，Leader 节点只有在收到更大 Term 值的消息时才会切换成 Follower 状态。故在发生网络分区时，即使在其他分区里新的 Leader 节点已经被选举出来，旧的 Leader 节点由于接收不到新 Leader 节点的心跳消息，依然会认为自己是当前集群的 Leader 节点（与其同一网络分区的 Follower 节点也认为它是当前集群的 Leader 节点），它依然会接收客户端的请求，但无法向客户端返回任何响应。CheckQuorum 机制的意思是：每隔一段时间，Leader 节点会尝试连接集群中的其他节点（发送心跳消息），如果发现自己可以连接到节点个数没有超过半数（即没有收到足够的心跳响应），则主动切换成 Follower 状态。这样，在上述网络分区的场景中，旧的 Leader 节点可以很快知道自己已经过期，可以减少 Client 连接旧 Leader 节点的等待时间。



仅供非商业用途或交流学习使用

- preVote（bool 类型）：在上一章介绍 Raft 协议时提到，Follower 节点在选举计时器超时之后，会切换成 Candidate 状态并发起选举。然而 Follower 节点超时没有收到心跳消息时，也可能是由于 Follower 节点自身的网络问题导致的，例如，前面提到的网络分区的场景。即使如此，该 Follower 节点还是会不断地发起选举，其 Term 值也会不断递增。待该 Follower 节点的网络故障恢复并收到 Leader 节点的心跳消息时，由于其 Term 值已经增加，该 Follower 节点会丢弃掉 Term 值比其自身小的心跳消息，之后就会触发一次没有必要进行的 Leader 选举。在前面介绍 Raft 协议时也提到了 PreVote 优化避免上述情况，当 Follower 节点准备发起一次选举之前，会先连接集群中的其他节点，并询问它们是否愿意参与选举，如果集群中的其他节点能够正常收到 Leader 节点的心跳消息，则会拒绝参与选举，反之则参与选举。当在 PreVote 过程中，有超过半数的节点响应并参与新一轮选举，则可以发起新一轮的选举。
- randomizedElectionTimeout(int 类型): 该字段是 electiontimeout～2×electiontimeout-1 之间的随机值，也是选举计时器的上限，当 electionElapsed 超过该值时即为超时。
- tick（func()类型）：当前节点推进逻辑时钟的函数。如果当前节点是 Leader，则指向 raft.tickHeartbeat() 函数，如果当前节点是 Follower 或是 Candidate，则指向 raft.tickElection()函数。
- step（stepFunc 类型）：当前节点收到消息时的处理函数。如果是 Leader 节点，则该字段指向 stepLeader()函数，如果是 Follower 节点，则该字段指向 stepFollower()函数，如果是处于 preVote 阶段的节点或是 Candidate 节点，则该字段指向 stepCandidate()函数。
- readStates（[]ReadState 类型）：与只读请求相关，后面会详细介绍只读请求的相关内容。
- readOnly(*readOnly 类型): 与只读请求相关，后面会详细介绍只读请求的相关内容。

在开始详细介绍 raft 提供的方法之前，我们需要先了解支持 raft 功能的相关组件。

3.1.1　Config结构体

Config 结构体主要用于配置参数的传递，在创建 raft 实例时需要的参数会通过 Config 实例传递进去。Config 的核心字段如下。

- ID（uint64 类型）：当前节点的 ID。
- peers（[]uint64 类型）：记录了集群中所有节点的 ID。
- ElectionTick（int 类型）：用于初始化 raft.electionTimeout，即逻辑时钟连续推进多少次



仅供非商业用途或交流学习使用

后，就会触发 Follower 节点的状态切换及新一轮的 Leader 选举。

- HeartbeatTick（int 类型）：用于初始化 raft.heartbeatTimeout，即逻辑时钟连续推进多少次后，就触发 Leader 节点发送心跳消息。
- Storage（Storage 类型）：当前节点保存 raft 日志记录使用的存储，后面会有单独的小节介绍 Storage 接口及其实现。
- Applied（uint64 类型）：当前已经应用的记录位置（已应用的最后一条 Entry 记录的索引值），该值在节点重启时需要设置，否则会重新应用已经应用过 Entry 记录。
- MaxSizePerMsg（uint64 类型）：用于初始化 raft.maxMsgSize 字段，每条消息的最大字节数，如果是 math.MaxUint64 则没有上限，如果是 0 则表示每条消息最多携带一条 Entry。
- MaxInflightMsgs（int 类型）：用于初始化 raft.maxInflight，即已经发送出去且未收到响应的最大消息个数。
- CheckQuorum（bool 类型）：是否开启 CheckQuorum 模式，用于初始化 raft.checkQuorum 字段，CheckQuorum 模式前面已经详细介绍过了，不再赘述。
- PreVote（bool 类型）：是否开启 PreVote 模式，用于初始化 raft.preVote 字段，PreVote 模式前面已经详细介绍过了，不再赘述。
- ReadOnlyOption（ReadOnlyOption 类型）：与只读请求的处理相关，后面详细介绍。

Config 的字段都是公开的，主要用于参数传递，所以没有特殊方法需要介绍。

3.1.2 Storage接口及其实现

在 etcd-raft 模块中定义了 Storage 接口，其主要作用就是存储当前节点接收到的 Entry 记录。下面来分析一下 Storage 接口的定义，具体实现如下：

```
type Storage interface {
    // 返回 Storage 中记录的状态信息，返回的是 HardState 实例和 ConfState 实例
    // 在前面介绍 Raft 协议时提到，集群中每个节点都需要保存一些必需的基本信息，在 etcd 中将其
    // 封装成 HardState，其中主要封装了当前任期号（Term 字段）、当前节点在该任期中将选票投
    // 给了哪个节点（Vote 字段）、已提交 Entry 记录的位置（Commit 字段，即最后一条已提交记录的索引值）
    // ConfState 中封装了当前集群中所有节点的 ID（Nodes 字段）
    InitialState() (pb.HardState, pb.ConfState, error)

    // 在 Storage 中记录了当前节点的所有 Entry 记录，Entries 方法返回指定范围的 Entry 记录（[lo,hi)），
    // 第三个参数（maxSize）限定了返回的 Entry 集合的字节数上限
```



仅供非商业用途或交流学习使用

```
    Entries(lo, hi, maxSize uint64) ([]pb.Entry, error)

    // 该方法返回 Storage 中记录的第一条 Entry 的索引值(Index)，在该 Entry 之前的所有 Entry 都已
    // 经被包含进了最近的一次 SnapShot 中
    FirstIndex() (uint64, error)

    // 该方法返回 Storage 中记录的第一条 Entry 的索引值(Index)
    LastIndex() (uint64, error)

    Term(i uint64) (uint64, error)   // 查询指定 Index 对应的 Entry 的 Term 值

    Snapshot() (pb.Snapshot, error) // 返回最近一次生成的快照数据
}
```

MemoryStorage 是 etcd-raft 模块为 Storage 接口提供的一个实现，从名字也能看出，MemoryStorage 在内存中维护上述状态信息（hardState 字段）、快照数据（snapshot 字段）及所有的 Entry 记录（ents 字段，[]raftpb.Entry 类型），在 MemoryStorage.ents 字段中维护了快照数据之后的所有 Entry 记录。另外需要注意的是，MemoryStorage 继承了 sync.Mutex，MemoryStorage 中的大部分操作是需要加锁同步的。通过这里的介绍，我们大概可以了解 MemoryStorage 的结构，如图 3-1 所示。

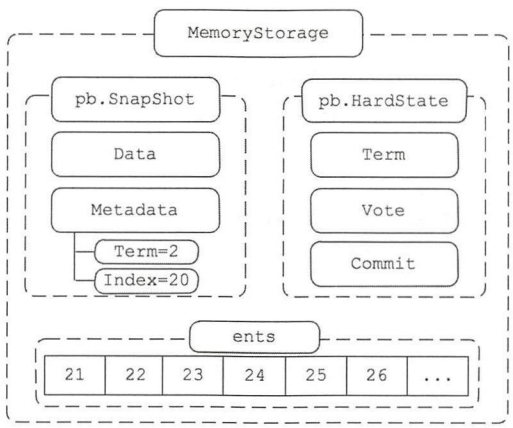


图 3-1

MemoryStorage.InitialState()方法会直接返回 hardState 字段中记录的 HardState 实例并使用快照的元数据中记录信息创建 ConfState 实例返回。MemoryStorage 的 FirstIndex()方法和 LastIndex()方法分别返回了 ents 数组中第一个元素的 Index 字段值和最后一个元素的 Index 字段值。这几个实现比较简单，这里不再展开介绍。



仅供非商业用途或交流学习使用

当 MemoryStorage 需要更新快照数据时，会调用 MemoryStorage.ApplySnapshot()方法将 SnapShot 实例保存到 MemoryStorage 中，例如，在节点重启时，就会通过读取快照文件创建对应的 SnapShot 实例，然后保存到 MemoryStorage 中。MemoryStorage.ApplySnapshot()方法的具体实现如下：

```
func (ms *MemoryStorage) ApplySnapshot(snap pb.Snapshot) error {
    ms.Lock() // 加锁同步，MemoryStorage 实现了 sync.Mutex
    defer ms.Unlock() // 方法结束后，释放锁

    // 通过快照的元数据比较当前 MemoryStorage 中记录的 SnapShot 与待处理的 SnapShot 数据的新旧程度
    msIndex := ms.snapshot.Metadata.Index
    snapIndex := snap.Metadata.Index
    if msIndex >= snapIndex { // 比较两个 pb.SnapShot 所包含的最后一条记录的 Index 值
        return ErrSnapOutOfDate // 如果待处理 SnapShot 数据比较旧，则直接抛出异常
    }

    ms.snapshot = snap // 更新 MemoryStorage.snapshot 字段
    // 重置 MemoryStorage.ents 字段，此时在 ents 中只有一个空的 Entry 实例
    ms.ents = []pb.Entry{{Term: snap.Metadata.Term, Index: snap.Metadata.Index}}
    return nil
}
```

设置完快照数据之后，就可以开始向 MemoryStorage 中追加 Entry 记录了，该功能主要由 MemoryStorage.Append()方法完成，其具体实现如下：

```
func (ms *MemoryStorage) Append(entries []pb.Entry) error {
    // 检测 entries 切片的长度(略)，加锁/解锁的逻辑(略)
    first := ms.firstIndex()// 获取当前 MemoryStorage 的 FirstIndex 值
    // 获取待添加的最后一条 Entry 的 Index 值
    last := entries[0].Index + uint64(len(entries)) - 1
    if last < first { // entries 切片中所有的 Entry 都已经过时，无须添加任何 Entry
        return nil
    }

    // first 之前的 Entry 已经记入 SnapShot 中，不应该再记录到 ents 中，所以将这部分 Entry 截掉
    if first > entries[0].Index {
        entries = entries[first-entries[0].Index:]
    }
```



仅供非商业用途或交流学习使用

```
    // 计算 entries 切片中第一条可用的 Entry 与 first 之间的差距
    offset := entries[0].Index - ms.ents[0].Index
    switch {
    case uint64(len(ms.ents)) > offset: // 这是图 3-2 所示的场景(A)
        // 保留 MemoryStorage.ents 中 first~offset 的部分，offset 之后的部分被抛弃
        ms.ents = append([]pb.Entry{}, ms.ents[:offset]...)
        // 然后将待追加的 Entry 追加到 MemoryStorage.ents 中
        ms.ents = append(ms.ents, entries...)
    case uint64(len(ms.ents)) == offset: // 这是图 3-2 所示的场景(B)
        // 直接将待追加的日志记录(entries)追加到 MemoryStorage 中
        ms.ents = append(ms.ents, entries...)
    default: // 异常处理(略)
    }
    return nil
}
```

MemoryStorage.Append()方法的大致原理如图 3-2 所示。

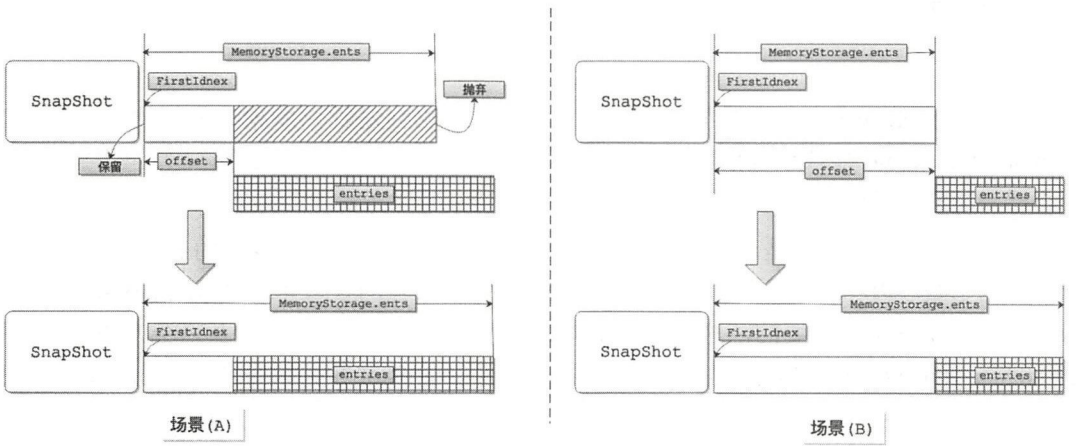


图 3-2

了解了 MemoryStorage 设置 SnapShot 和添加 Entry 的逻辑之后，下面介绍查询的相关方法。首先是 MemoryStorage.Entries()方法，它负责查询指定范围的 Entry，具体实现如下：

```
func (ms *MemoryStorage) Entries(lo, hi, maxSize uint64) ([]pb.Entry, error) {
    // 加锁和解锁的相关操作(略)
    // 如果待查询的最小 Index 值(参数 lo)小于 FirstIndex，则直接抛出异常(略)
```



仅供非商业用途或交流学习使用

```
    // 如果待查询的最大 Index 值(参数 hi)大于 LastIndex，则直接抛出异常(略)
    // 如果 MemoryStorage.ents 只包含一条 Entry，则其为空 Entry，直接抛出异常(略)
    ents := ms.ents[lo-offset: hi-offset] // 获取 lo~hi 之间的 Entry，并返回
    return limitSize(ents, maxSize), nil  // 限制返回 Entry 切片的总字节大小
}

// 下面是 limitSize()方法的具体实现：
func limitSize(ents []pb.Entry, maxSize uint64) []pb.Entry {
    // 检测 ents 切片的长度(略)
    size := ents[0].Size()
    var limit int
    // 遍历 ents 切片，查找 limit(limit 下标之前所有 Entry 的字节数之和，小于 maxSize 值)
    for limit = 1; limit < len(ents); limit++ {
        size += ents[limit].Size()
        if uint64(size) > maxSize {
            break
        }
    }
    return ents[:limit] // 返回 ents 切片中 limit 之前的部分
}
```

MemoryStorage.Term()方法与 Entries()方法类似，也会进行一系列边界检测，最终通过 MemoryStorage.ents 字段读取指定 Entry 的 Term 值，该方法的实现比较简单，这里不再展开介绍，感兴趣的读者可以参考源码。

最后需要读者了解的是，随着系统的运行，MemoryStorage.ents 中保存的 Entry 记录会不断增加，为了减小内存的压力，定期创建快照来记录当前节点的状态并压缩 MemoryStorage.ents 数组的空间是非常有必要的，这样就可以降低内存使用。这个过程中涉及三个方法，首先是 CreateSnapshot()方法，它会接收当前集群状态，以及 SnapShot 相关数据来更新 snapshot 字段，具体实现如下：

```
// 简单说明该方法的参数：i 是新建 SnapShot 包含的最大的索引值，cs 是当前集群的状态，
// data 是新建 SnapShot 的具体数据
func (ms *MemoryStorage) CreateSnapshot(i uint64, cs *pb.ConfState, data []byte)
        (pb.Snapshot, error) {
    // 加锁、解锁的相关代码(略)
    // 边界检查，i 必须大于当前 SnapShot 包含的最大 Index 值，
    // 并且小于 MemoryStorage 的 LastIndex 值，否则抛出异常(略)
```

```
    offset := ms.ents[0].Index
    // 更新 MemoryStorage.snapshot 的元数据
    ms.snapshot.Metadata.Index = i
    ms.snapshot.Metadata.Term = ms.ents[i-offset].Term
    if cs != nil {
      ms.snapshot.Metadata.ConfState = *cs
    }
    ms.snapshot.Data = data // 更新具体的快照数据
    return ms.snapshot, nil
}
```

新建 SnapShot 之后，一般会调用 MemoryStorage.Compact()方法将 MemoryStorage.ents 中指定索引之前的 Entry 记录全部抛弃，从而实现压缩 MemoryStorage.ents 的目的，具体实现如下：

```
func (ms *MemoryStorage) Compact(compactIndex uint64) error {
    // 加锁、解锁的操作(略)
    offset := ms.ents[0].Index
    // 边界检测，firstIndex < compactIndex < lastIndex(略)
    i := compactIndex - offset // compactIndex 对应 Entry 的下标
    // 创建新的切片，用来存储 compactIndex 之后的 Entry
    ents := make([]pb.Entry, 1, 1+uint64(len(ms.ents))-i)
    ents[0].Index = ms.ents[i].Index
    ents[0].Term = ms.ents[i].Term
    // 将 compactIndex 之后的 Entry 拷贝到 ents 中，并更新 MemoryStorage.ents 字段
    ents = append(ents, ms.ents[i+1:]...)
    ms.ents = ents
    return nil
}
```

最后，上层模块可以通过 MemoryStorage.Snapshot()方法获取 SnapShot，该方法直接返回 MemoryStorage.snapshot 字段，实现比较简单，不再展开介绍。

3.1.3　unstable结构体

在 etcd-raft 模块中，除了上面介绍的 Storage，还有一个存储 Entry 记录的地方，就是本小节要介绍的 unstable 结构体。unstable 使用内存数组维护其中所有的 Entry 记录，对于 Leader 节点而言，它维护了客户端请求对应的 Entry 记录；对于 Follower 节点而言，它维护的是从 Leader

节点复制来的 Entry 记录。无论是 Leader 节点还是 Follower 节点，对于刚刚接收到的 Entry 记录首先都会被存储在 unstable 中。然后按照 Raft 协议将 unstable 中缓存的这些 Entry 记录交给上层模块进行处理，上层模块会将这些 Entry 记录发送到集群其他节点或进行保存（写入 Storage 中）。之后，上层模块会调用 Advance()方法通知底层的 etcd-raft 模块将 unstable 中对应的 Entry 记录删除（因为已经保存到了 Storage 中）。正因为 unstable 中保存的 Entry 记录并未进行持久化，可能会因节点故障而意外丢失，所以被称为 unstable。

通过上面的描述，大致可以得到下面这张结构，如图 3-3 所示。

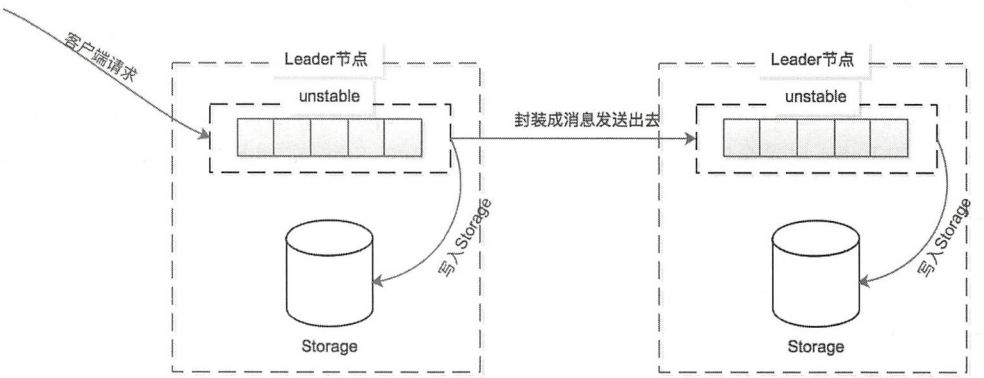


图 3-3

下面介绍一下 unstable 中的核心字段。

- **entries（[]pb.Entry 类型）**：用于保存未写入 Storage 中的 Entry 记录。
- **offset（uint64 类型）**：entries 中的第一条 Entry 记录的索引值。
- **snapshot（*pb.Snapshot 类型）**：快照数据，该快照数据也是未写入 Storage 中的。

在 unstable 中提供了很多与 Storage 类似的方法，在下一小节介绍的 raftLog 中，很多方法都是先尝试调用 unstable 的相应方法，在其失败后（unstable 的方法返回（0，false）即表示失败），再尝试调用 Storage 的对应方法。

unstable.maybeFirstIndex()方法会尝试获取 unstable 的第一条 Entry 记录的索引值，unstable.maybeLastIndex()方法会尝试获取 unstable 的最后一条 Entry 记录的索引值，如果获取失败则返回（0，false），两者的具体实现如下：

```
func (u *unstable) maybeFirstIndex() (uint64, bool) {
    // 如果 unstable 记录了快照，则通过快照元数据返回相应的索引值
    // 读者可能觉得该方法的实现出乎意料，并认为 maybeFirstIndex()方法应该
    // 与 MemoryStorage.firstIndex()方法类似，返回 unstable.entries 中第一条 Entry 记录的索引值
    // 后面介绍 raftLog.firstIndex()方法时，还会提及该方法，读者此处先了解其实现即可
```

```
    if u.snapshot != nil {
        return u.snapshot.Metadata.Index + 1, true
    }
    return 0, false
}

func (u *unstable) maybeLastIndex() (uint64, bool) {
    if l := len(u.entries); l != 0 { // 检测 entries 切片的长度
        return u.offset + uint64(l) - 1, true // 返回 entries 中最后一条 Entry 记录的索引值
    }
    if u.snapshot != nil { // 如果存在快照数据，则通过其元数据返回索引值
        return u.snapshot.Metadata.Index, true
    }
    return 0, false // entries 和 snapshot 都是空的，则整个 unstable 其实也就没有任何数据了
}
```

unstable.maybeTerm()方法的主要功能是尝试获取指定 Entry 记录的 Term 值，根据条件查找指定的 Entry 记录的位置，大致如图 3-4 所示。

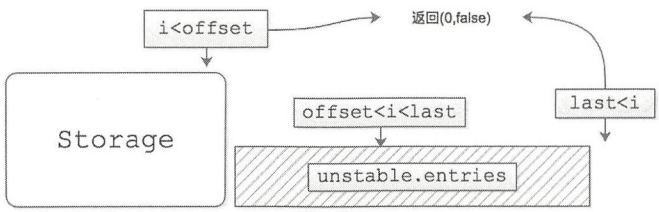


图 3-4

unstable.maybeTerm()方法的具体实现如下：

```
func (u *unstable) maybeTerm(i uint64) (uint64, bool) {
    // 指定索引值对应的 Entry 记录已经不在 unstable 中，可能已经被持久化或是被写入快照
    if i < u.offset {
        if u.snapshot == nil {
            return 0, false
        }
        if u.snapshot.Metadata.Index == i { // 检测是不是快照所包含的最后一条 Entry 记录
            return u.snapshot.Metadata.Term, true
        }
        return 0, false
    }
```

仅供非商业用途或交流学习使用


```
    last, ok := u.maybeLastIndex() // 获取 unstable 中最后一条 Entry 记录的索引
    if !ok {
        return 0, false
    }
    if i > last { // 指定的索引值超出了 unstable 已知范围，查找失败
        return 0, false
    }
    return u.entries[i-u.offset].Term, true    // 从 entries 字段中查找指定的 Entry 并返回
                                               // 其 Term 值
}
```

当 unstable.entries 中的 Entry 记录已经被写入 Storage 之后，会调用 unstable.stableTo()方法清除 entries 中对应的 Entry 记录，stableTo()方法的具体实现如下：

```
func (u *unstable) stableTo(i, t uint64) {
    // 查找指定 Entry 记录的 Term 值，若查找失败则表示对应的 Entry 不在 unstable 中，直接返回(略)
    gt, ok := u.maybeTerm(i)
    if gt == t && i >= u.offset { // 指定的 Entry 记录在 unstable.entries 中保存
        // 指定索引值之前的 Entry 记录都已经完成持久化，则将其之前的全部 Entry 记录删除
        u.entries = u.entries[i+1-u.offset:]
        u.offset = i + 1 // 更新 offset 字段
        // 随着多次追加日志和截断日志的操作，unstable.entires 底层的数组会越来越大，
        // shrinkEntriesArray()方法会在底层数组长度超过实际占用的两倍时，对底层数组进行缩减
        u.shrinkEntriesArray()
    }
}

// 下面是 shrinkEntriesArray()方法的具体实现：
func (u *unstable) shrinkEntriesArray() {
    const lenMultiple = 2
    if len(u.entries) == 0 { // 检测 entries 的长度
        u.entries = nil
    } else if len(u.entries)*lenMultiple < cap(u.entries) {
        newEntries := make([]pb.Entry, len(u.entries)) // 重新创建切片
        copy(newEntries, u.entries) // 复制原有切片中的数据
        u.entries = newEntries // 重置 entries 字段
    }
}
```

仅供非商业用途或交流学习使用

同理，当unstable.snapshot字段指向的快照被写入Storage之后，会调用unstable.stableSnapTo()方法将 snapshot 字段清空，实现比较简单，代码就不贴出来了。

unstable.truncateAndAppend()方法的主要功能是向 unstable.entries 中追加 Entry 记录，其实现与 Storage.Append()方法类似，也会涉及截断的场景。通过图 3-5 可以清晰地了解truncateAndAppend()方法完成的操作。

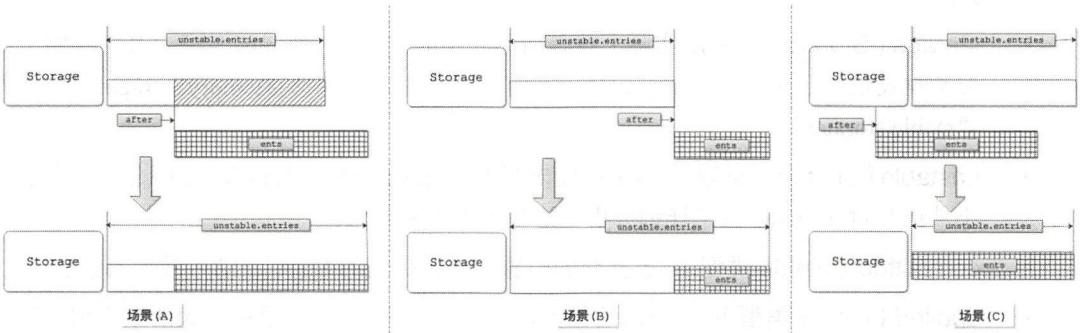


图 3-5

unstable.truncateAndAppend()方法的具体实现如下：

```
func (u *unstable) truncateAndAppend(ents []pb.Entry) {
    after := ents[0].Index // 获取第一条待追加的 Entry 记录的索引值
    switch {
    case after == u.offset+uint64(len(u.entries)):
        // 若待追加的记录与 entries 中的记录正好连续，则可以直接向 entries 中追加
        u.entries = append(u.entries, ents...)
    case after <= u.offset: // 直接用待追加的 Entry 记录替换当前的 entries 字段，并更新 offset
        u.offset = after
        u.entries = ents
    default:
        // after 在 offset~last 之间，则 after~last 之间的 Entry 记录冲突。这里会将 offset~after
        // 之间的记录保留，抛弃 after 之后的记录，然后完成追加操作
        // unstable.slice()方法会检测 after 是否合法，并返回 offset~after 的切片，其具体实现(略)
        u.entries = append([]pb.Entry{}, u.slice(u.offset, after)...)
        u.entries = append(u.entries, ents...)
    }
}
```

仅供非商业用途或交流学习使用

到此为止，unstable 的核心字段和方法都已经介绍完了。

3.1.4 raftLog结构体

Raft 协议中日志复制部分的核心就是在集群中各个节点之间完成日志的复制，因此在 etcd-raft 模块的实现中使用 raftLog 结构来管理节点上的日志，它依赖于前面介绍的 Storage 接口和 unstable 结构体。下面来看一下 raftLog 中核心字段的含义与功能。

- **storage (Storage 类型)**：实际上就是前面介绍的 MemoryStorage 实例，其中存储了快照数据及该快照之后的 Entry 记录。在有的文档中，也将该 MemoryStorage 实例称为“stable storage”。
- **unstable (unstable 类型)**：用于存储未写入 Storage 的快照数据及 Entry 记录，在上一小节已经对 unstable 做了详细分析，这里不再重复介绍。
- **committed (uint64 类型)**：已提交的位置，即已提交的 Entry 记录中最大的索引值。
- **applied (uint64 类型)**：已应用的位置，即已应用的 Entry 记录中最大的索引值。其中 committed 和 applied 之间始终满足 committed≤applied 这个不等式关系。

了解了 raftLog 中核心字段的含义之后，我们来看一下 newLog()函数是如何创建一个 raftLog 实例的，具体实现如下：

```
func newLog(storage Storage, logger Logger) *raftLog {
  log := &raftLog{ // 创建 raftLog 实例，并初始化 storage 字段
    storage: storage,
    logger: logger,
  }
  // 获取 Storage 中的第一条 Entry 和最后一条 Entry 的索引值，异常处理(略)
  firstIndex, err := storage.FirstIndex()
  lastIndex, err := storage.LastIndex()
  log.unstable.offset = lastIndex + 1 // 初始化 unstable.offset
  // 初始化 committed、applied 字段
  log.committed = firstIndex - 1
  log.applied = firstIndex - 1
  return log
}
```

通过 newLog()函数的创建过程，我们可以得到图 3-6。

仅供非商业用途或交流学习使用

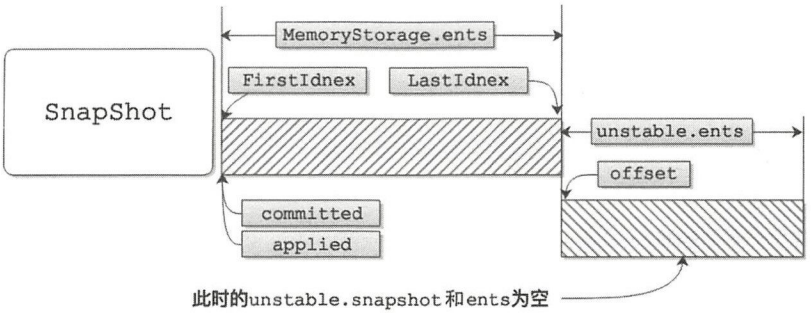


图 3-6

当 Follower 节点或 Candidate 节点需要向 raftLog 中追加 Entry 记录时，会通过 raft.handleAppendEntries()方法调用 raftLog.maybeAppend()方法完成追加 Entry 记录的功能。

raftLog.maybeAppend()方法的大致步骤如下。

（1）maybeAppend()方法的参数：第一个参数 index 是 MsgApp 消息携带的第一条 Entry 的 Index 值，这里的 MsgApp 消息是 etcd-raft 模块中定义的消息类型之一，对应 Raft 协议中提到的 Append Entries 消息；第二个参数 logTerm 是 MsgApp 消息的 LogTerm 字段，通过消息中携带的该 Term 值与当前节点记录的 Term 进行比较，就可以判断消息是否为过时的消息；第三个参数 committed 是 MsgApp 消息的 Commit 字段，Leader 节点通过该字段通知 Follower 节点当前已提交 Entry 的位置；第四个参数 ents 是 MsgApp 消息中携带的 Entry 记录，即待追加到 raftLog 中的 Entry 记录。

（2）maybeAppend()方法首先会调用 matchTerm()方法检测 MsgApp 消息的 Index 字段及 LogTerm 字段是否合法，raftLog.matchTerm()方法的实现如下：

```
func (l *raftLog) matchTerm(i, term uint64) bool {
    t, err := l.term(i) // 查询指定索引值对应的 Entry 记录的 Term 值，异常处理(略)
    return t == term
}
```

raftLog.term()方法先会去 unstable 中查找相应的 Entry 记录，如果查找不到，则再去 storage 中查找，term()方法的具体实现如下：

```
func (l *raftLog) term(i uint64) (uint64, error) {
    // 检测指定的索引值的合法性(略)
    // 尝试从 unstable 中获取对应的 Entry 记录并返回其 Term 值
    if t, ok := l.unstable.maybeTerm(i); ok {
        return t, nil
```

仅供非商业用途或交流学习使用


```
    }
    // 尝试从 storage 中获取对应的 Entry 记录并返回其 Term 值
    t, err := l.storage.Term(i)
    if err == nil {
        return t, nil
    }
    // 异常场景返回 0(略)
}
```

其中，MemoryStorage.Term()方法和 unstable.maybeTerm()方法在前面已经介绍过了，这里不再重复介绍。

（3）遍历待追加的 Entry 集合，查找是否与 raftLog 中已有的 Entry 发生冲突（Index 相同但 Term 不同），该过程在 raftLog.findConflict()方法中完成，具体实现如下：

```
func (l *raftLog) findConflict(ents []pb.Entry) uint64 {
    // 遍历全部待追加的 Entry，判断 raftLog 中是否存在冲突的 Entry 记录
    for _, ne := range ents {
        if !l.matchTerm(ne.Index, ne.Term) { // 查找冲突的 Entry 记录
            return ne.Index // 返回冲突记录的索引值
        }
    }
    return 0 // 如果没有发生冲突 Entry，则返回 0
}
```

需要注意 findConflict()方法的返回值：当待追加的 Entry 记录在 raftLog 中不存在时，会返回第一条不存在 Entry 记录的索引值；raftLog 包含全部的待追加记录且没有发生冲突，则返回 0，如图 3-7 所示。

仅供非商业用途或交流学习使用

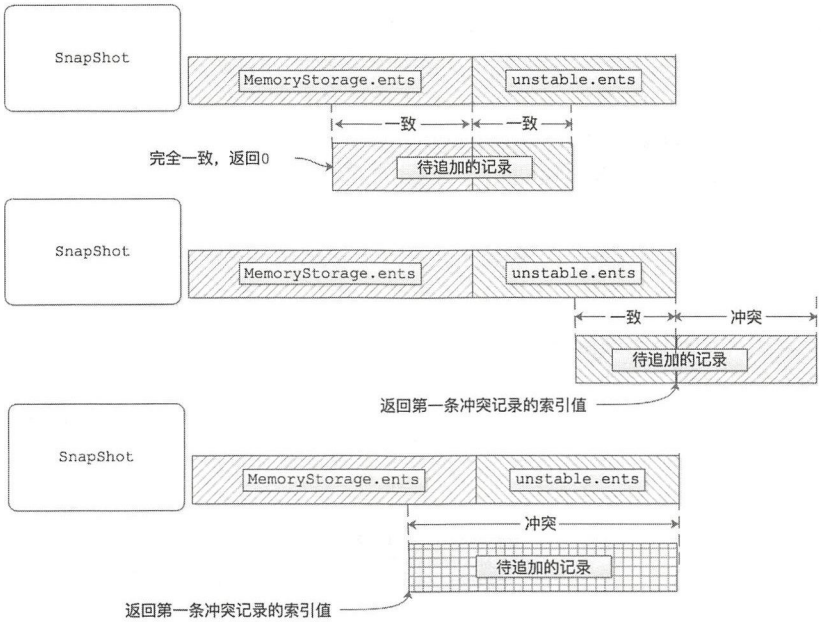


图 3-7

（4）raftLog 会根据 findConflict()方法的返回值决定如何进行后续的追加操作，相关的代码片段如下：

```
switch {
case ci == 0: // findConflict()方法返回 0 时，表示 raftLog 中已经包含了所有待追加
              // 的 Entry 记录，不必进行任何追加操作
case ci <= l.committed: // 如果出现冲突的位置是已提交的记录，则输出异常日志并终止整个程序
   l.logger.Panicf("...", ci, l.committed)
default: // 如果冲突位置是未提交的部分
   offset := index + 1
   l.append(ents[ci-offset:]...) // 则将 ents 中未发生冲突的部分追加到 raftLog 中
}
```

raftLog.append()方法主要通过调用 unstable.truncateAndAppend()方法完成记录的追加功能，其实现如下：

```
func (l *raftLog) append(ents ...pb.Entry) uint64 {
   // 合法性检测(略)
   // 调用 unstable.truncateAndAppend()方法将 Entry 记录追加到 unstable 中
   l.unstable.truncateAndAppend(ents)
```

仅供非商业用途或交流学习使用


```
    return l.lastIndex() // 返回 raftLog 最后一条日志记录的索引
}
```

（5）通过 commitTo()方法更新 raftLog.committed 字段，图 3-8 中列举了两种更新 commitTo 字段的场景，其他场景留给读者考虑。

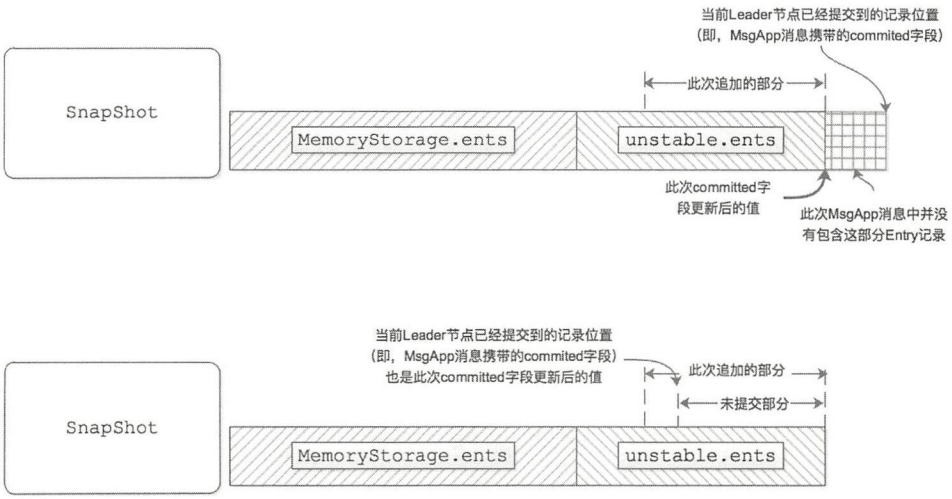


图 3-8

raftLog.commitTo()方法的具体实现如下：

```
func (l *raftLog) commitTo(tocommit uint64) {
  if l.committed < tocommit { // raftLog.committed 字段只能后移，不能前移
    if l.lastIndex() < tocommit {
      l.logger.Panicf("...")
    }
    l.committed = tocommit // 更新 committed 字段
  }
}
```

了解了具体每一步的实现之后，将它们串起来，就是 raftLog.maybeAppend()方法的大致逻辑，其具体实现就不再粘贴出来了，读者可以参考上面的分析来查看相应的代码。

raftLog.firstIndex()和 lastIndex()方法与前面介绍的 MemoryStorage 的 firstIndex()和 lastIndex()方法类似，它们返回的是 raftLog 中第一条和最后一条 Entry 记录的索引值。这两个方法分别先尝试通过 unstable.maybeFirstIndex ()和 maybeLastIndex()方法进行查找，如果查找失败，则通过 Storage.FirstIndex()和 LastIndex()方法进行查找。raftLog.firstIndex()和 lastIndex()方法的实现比较

仅供非商业用途或交流学习使用

简单，代码就不再粘贴了，感兴趣的读者请参考源码进行学习。

当上层模块需要从 raftLog 获取 Entry 记录进行处理时，会先调用 hasNextEnts()方法检测是否有待应用的记录，然后调用 nextEnts()方法将已提交且未应用的 Entry 记录返回给上层模块处理。这两个方法的具体实现如下：

```
func (l *raftLog) nextEnts() (ents []pb.Entry) {
    off := max(l.applied+1, l.firstIndex()) // 获取当前已经应用记录的位置
    if l.committed+1 > off { // 是否存在已提交且未应用的 Entry 记录
        // 获取全部已提交且未应用的 Entry 记录并返回，异常处理(略)
        ents, err := l.slice(off, l.committed+1, noLimit)
        return ents
    }
    return nil
}

// 下面是 hasNextEnts()方法的具体实现
func (l *raftLog) hasNextEnts() bool {
    off := max(l.applied+1, l.firstIndex()) // 获取当前已经应用记录的位置
    return l.committed+1 > off // 是否存在已提交且未应用的 Entry 记录
}
```

在 Leader 节点向 Follower 节点发送 MsgApp 消息时，需要根据 Follower 节点的日志复制（NextIndex 和 MatchIndex）情况决定发送的 Entry 记录，此时需要调用 raftLog.entries()方法获取指定的 Entry 记录，该方法的具体实现如下：

```
func (l *raftLog) entries(i, maxsize uint64) ([]pb.Entry, error) {
    // 检测指定的 Entry 记录索引值是否超出了 raftLog 的范围(略)
    return l.slice(i, l.lastIndex()+1, maxsize)   // 获取指定范围的 Entry 记录
}

func (l *raftLog) slice(lo, hi, maxSize uint64) ([]pb.Entry, error) {
    // 边界检测(略)，检测的条件是 l.firstIndex() <= lo < hi <= l.lastIndex()
    // 如果 lo 小于 unstable.offset(unstable 中第一条记录的索引)，则需要从 raftLog.storage 中
    // 获取记录
    if lo < l.unstable.offset {
        // 从 Storage 中获取记录，可能只能从 Storage 中取前半部分(也就是到 lo~l.unstable.offset
        // 部分)
        // 注意 maxSize 参数，它会限制获取的记录切片的总字节数
```

仅供非商业用途或交流学习使用


```
        storedEnts, err := l.storage.Entries(lo, min(hi, l.unstable.offset), maxSize)
        // 异常处理(略)
        // 从 Storage 中取出的记录已经达到了 maxSize 的上限，则直接返回
        if uint64(len(storedEnts)) < min(hi, l.unstable.offset)-lo {
            return storedEnts, nil
        }
        ents = storedEnts
    }

    // 从 unstable 中取出后半部分记录，即 l.unstable.offset~hi 的日志记录
    if hi > l.unstable.offset {
        unstable := l.unstable.slice(max(lo, l.unstable.offset), hi)
        // 将 Storage 中获取的记录与 unstable 中获取的记录进行合并
        if len(ents) > 0 {
            ents = append([]pb.Entry{}, ents...)
            ents = append(ents, unstable...)
        } else {
            ents = unstable
        }
    }
    // 限制获取的记录切片的总字节数，该方法在 Storage 小节已经详细介绍过了，这里不再赘述
    return limitSize(ents, maxSize), nil
}
```

图 3-9 展示了 raftLog.slice()方法查找 Entry 记录的几种场景，其他的场景留给读者思考。

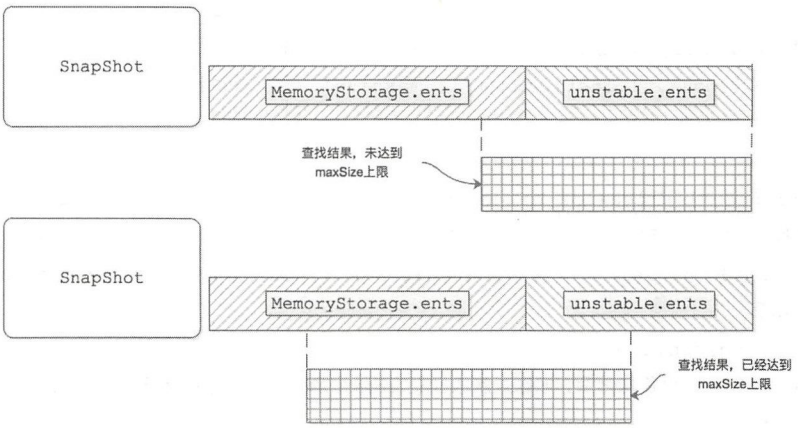


图 3-9

仅供非商业用途或交流学习使用

在前面介绍 Raft 协议时提到过，Follower 节点在接收到 Candidate 节点的选举请求之后，会通过比较 Candidate 节点的本地日志与自身本地日志的新旧程度，从而决定是否投票。raftLog 提供了 isUpToDate()方法用于比较日志的新旧程度，具体实现如下：

```
// 首先注意一下参数，lasti 和 term 分别是 Candidate 节点的最大记录索引值和最大任期号(即 MsgVote
// 请求(Candidate 发送的选举请求)携带的 Index 和 LogTerm)
func (l *raftLog) isUpToDate(lasti, term uint64) bool {
   // 比较日志新旧的方式也如前面介绍的一样，先比较任期号，任期号相同时再比较索引值
   return term > l.lastTerm() || (term == l.lastTerm() && lasti >= l.lastIndex())
}
```

到此为止，raftLog 中核心方法的具体实现就介绍完了，raftLog 中还有几个实现比较简单的方法，请读者参考源码分析。

3.1.5　raft实现

在后面的介绍中我们可以看到，etcd-raft 模块中只实现了 Raft 协议的基本行为，并未实现发送消息的功能（即网络层的相关内容），而是将网络层实现独立成了一个单独的模块，也就是下一章要介绍的 etcd-rafthttp 模块。在本节的前半部分详细介绍了 raft 中使用到的组件，本小节将详细介绍 raft 中的方法，也是 etcd-raft 模块的核心实现。

1. 初始化

下面先来看 newRaft()函数，它会根据传入的 Config 实例中携带的参数创建 raft 实例并初始化 raft 使用到的相关组件，具体实现如下：

```
func newRaft(c *Config) *raft {
   // 检测参数 Config 中各字段的合法性，异常处理(略)
   // 创建 raftLog 实例，用于记录 Entry 记录
   raftlog := newLog(c.Storage, c.Logger)
   // 获取 raftLog.storage 的初始状态(HardState 和 ConfState)，异常处理(略)
   // Storage 的初始状态是通过本地 Entry 记录回放得到的
   hs, cs, err := c.Storage.InitialState()
   // 根据快照中记录的集群信息和 Config 的配置信息初始化集群的节点信息(略)

   // 创建 raft 实例，其中字段的含义在本节开始处已经介绍过了，这里不再赘述
   r := &raft{
      id:              c.ID,   // 当前节点的 ID
```

仅供非商业用途或交流学习使用


```
        lead:            None,    // 当前集群中 Leader 节点的 ID，初始化时先被设置成 0
        raftLog:         raftlog,          // 负责管理 Entry 记录的 raftLog 实例
        // 每条消息的最大字节数，如果是 math.MaxUint64 则没有上限，如果是 0 则表示每条消息最多携带
        // 一条 Entry
        maxMsgSize:      c.MaxSizePerMsg,
        maxInflight:     c.MaxInflightMsgs, // 已经发送出去且未收到响应的最大消息个数
        prs:             make(map[uint64]*Progress),
        electionTimeout: c.ElectionTick,     // 选举超时时间
        heartbeatTimeout: c.HeartbeatTick,   // 心跳超时时间
        logger:          c.Logger,           // 普通日志输出
        checkQuorum:     c.CheckQuorum,      // 是否开始 CheckQuorum 模式
        preVote:         c.PreVote,          // 是否开启 PreVote 模式
        readOnly:        newReadOnly(c.ReadOnlyOption), // 只读请求的相关配置
    }

    // 初始化 raft.prs 字段，这里会根据集群中节点的 ID，为每个节点初始化 Progress 实例，
    // 在 Progress 中维护了对应节点的 NextIndex 值和 MatchIndex 值，以及一些其他的 Follower 节点信息
    // 注意：只有 Leader 节点的 raft.prs 字段是有效的
    for _, p := range peers {
        r.prs[p] = &Progress{Next: 1, ins: newInflights(r.maxInflight)}
    }
    // 根据从 Storage 中获取的 HardState，初始化 raftLog.committed 字段，以及 raft.Term 和 Vote 字段
    if !isHardStateEqual(hs, emptyState) {
        r.loadState(hs)
    }

    // 如果 Config 中配置了 Applied，则将 raftLog.applied 字段重置为指定的 Applied 值
    // 上层模块自己的控制正确的已应用位置时使用该配置
    if c.Applied > 0 {
        raftlog.appliedTo(c.Applied)
    }
    r.becomeFollower(r.Term, None) // 当前节点切换成 Follower 状态
    return r
}
```

2. 状态切换

raft 在特定的场景下（前面介绍 Raft 协议时已提到过这些场景，例如，选举计时器超时时，Follower 节点切换到 PreCandidate 状态等）会通过调用 become*()方法切换成指定状态的节点，

仅供非商业用途或交流学习使用

状态转换如图 3-10 所示。

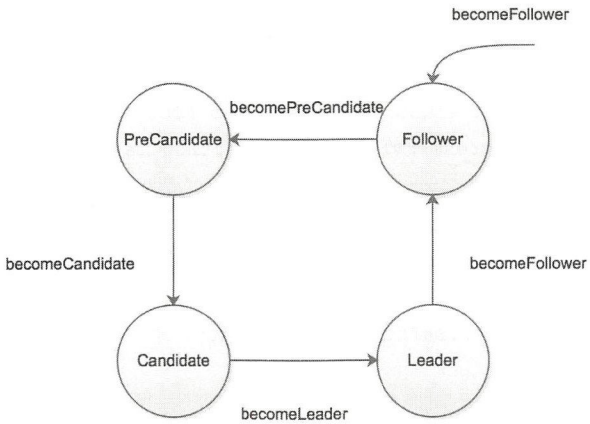


图 3-10

在 newRaft()函数中完成初始化之后，会调用 becomeFollower()方法将节点切换成 Follower 状态，其中会设置 raft 实例的多个字段，具体实现如下：

```
func (r *raft) becomeFollower(term uint64, lead uint64) {
    // 将 step 字段设置成 stepFollower，stepFollower()函数中封装了 Follower 节点处理消息的行为
    r.step = stepFollower
    r.reset(term)              // 重置 raft 实例 Term、Vote 等字段，后面详细介绍该方法
    r.tick = r.tickElection // 将 tick 字段设置成 tickElection 函数
    r.lead = lead               // 设置当前集群的 Leader 节点
    r.state = StateFollower // 设置当前节点的角色
}
```

除了 becomeFollower()方法，becomeLeader()和 becomeCandidate()方法都会调用 reset()方法，在 reset()方法中会重置 raft 实例的多个字段，其具体实现如下：

```
func (r *raft) reset(term uint64) {
    if r.Term != term {
        r.Term = term // 重置 Term 字段
        r.Vote = None // 重置 Vote 字段
    }
    r.lead = None // 清空 lead 字段

    // 重置选举计时器和心跳计时器
```

仅供非商业用途或交流学习使用


```
    r.electionElapsed = 0
    r.heartbeatElapsed = 0
    r.resetRandomizedElectionTimeout() // 重置选举计时器的过期时间(随机值)
    r.abortLeaderTransfer() // 清空 leadTransferee
    r.votes = make(map[uint64]bool) // 重置 votes 字段
    // 重置 prs，其中每个 Progress 中的 Next 设置为 raftLog.lastIndex
    for id := range r.prs {
        r.prs[id] = &Progress{Next: r.raftLog.lastIndex() + 1,
            ins: newInflights(r.maxInflight)}
        if id == r.id { // 将当前节点对应的 prs.Match 设置成 lastIndex
            r.prs[id].Match = r.raftLog.lastIndex()
        }
    }
    r.pendingConf = false // 清空 pendingConf 字段
    r.readOnly = newReadOnly(r.readOnly.option) // 只读请求的相关设置
}
```

当节点变成 Follower 状态之后，会周期性地调用 raft.tickElection()方法推进 electionElapsed 并检测是否超时，具体实现如下：

```
func (r *raft) tickElection() {
    r.electionElapsed++ // 递增 electionElapsed
    // promotable()方法会检查 prs 字段中是否还存在当前节点对应的 Progress 实例，这是为了检测当前
    // 节点是否被从集群中移除了
    // pastElectionTimeout()方法是检测当前的选举计时器是否超时(r.electionElapsed >=
    //  r.randomizedElectionTimeout)
    if r.promotable() && r.pastElectionTimeout() {
        r.electionElapsed = 0 // 重置 electionElapsed
        r.Step(pb.Message{From: r.id, Type: pb.MsgHup}) // 发起选举，后面详细介绍
    }
}
```

如果当前集群开启了 PreVote 模式，当 Follower 节点的选举计时器超时时，会先调用 becomePreCandidate()方法切换到 PreCandidate 状态，becomePreCandidate()方法的具体实现如下：

```
func (r *raft) becomePreCandidate() {
    // 检测当前节点的状态，禁止直接从 Leader 状态切换到 PreCandidate 状态(略)
    // 将 step 字段设置成 stepCandidate，stepCandidate()函数中封装了 PreCandidate 节点处理消
    // 息的行为
```

仅供非商业用途或交流学习使用


```
    r.step = stepCandidate
    r.tick = r.tickElection     // raft.tickElection()方法在前面介绍过，不再赘述
    r.state = StatePreCandidate // 修改当前节点的角色
}
```

当节点可以连接到集群中半数以上的节点时，会调用 becomeCandidate()方法切换到 Candidate 状态，becomeCandidate()方法的具体实现如下：

```
func (r *raft) becomeCandidate() {
    // 检测当前节点的状态，禁止直接从 Leader 状态切换到 PreCandidate 状态(略)
    // 将 step 字段设置成 stepCandidate，stepCandidate()函数中封装了 Candidate 节点处理消息的行为
    r.step = stepCandidate
    // 重置 raft 实例 Term、Vote 等字段，注意，这里的 Term 已经递增
    // reset()方法前面介绍过了，这里不再赘述
    r.reset(r.Term + 1)
    r.tick = r.tickElection  // raft.tickElection()方法在前面介绍过，不再赘述
    r.Vote = r.id            // 在此次选举中，Candidate 节点会将选票投给自己
    r.state = StateCandidate // 修改当前节点的角色
    // 注意：切换成 Candidate 状态之后，raft.Leader 字段为 None，这与其他状态不一样
}
```

当 Candidate 节点得到集群中半数以上节点的选票时，会调用 becomeLeader()方法切换成 Leader 状态，becomeLeader()方法的具体实现如下：

```
func (r *raft) becomeLeader() {
    // 检测当前节点的状态，禁止从 Follower 状态切换成 Leader 状态(略)
    // 将 step 字段设置成 stepLeader，stepLeader()函数中封装了 Leader 节点处理消息的行为
    r.step = stepLeader
    r.reset(r.Term)             // 重置 raft 实例 Term、Vote 等字段
    r.tick = r.tickHeartbeat    // 将 tick 字段设置成 tickHeartbeat 函数
    r.lead = r.id               // 将 leader 字段当前节点的 id
    r.state = StateLeader       // 更新当前节点的角色

    // 获取当前节点中所有未提交的 Entry 记录，异常处理(略)
    ents, err := r.raftLog.entries(r.raftLog.committed+1, noLimit)
    // 检测未提交的记录中是否存在多条集群配置变更的 Entry 记录(即 EntryConfChange 类型的 Entry 记录)
    // 如果存在多条 EntryConfChange 类型记录，则异常关闭(略)
    if nconf == 1 { // 如果只有一条，则将 pendingConf 设置为 true
```

仅供非商业用途或交流学习使用


```
        r.pendingConf = true
    }

    // 向当前节点的 raftLog 中追加一条空的 Entry 记录
    r.appendEntry(pb.Entry{Data: nil})
}
```

这里重点看一下 raft.appendEntry()方法，它的主要操作步骤如下：

（1）设置待追加的 Entry 记录的 Term 值和 Index 值。

（2）向当前节点的 raftLog 中追加 Entry 记录。

（3）更新当前节点对应的 Progress 实例。

（4）尝试提交 Entry 记录，即修改 raftLog.committed 字段的值。

raft.appendEntry()方法的具体实现如下：

```
func (r *raft) appendEntry(es ...pb.Entry) {
    li := r.raftLog.lastIndex() // 获取 raftLog 中最后一条记录的索引值
    for i := range es { // 更新待追加记录的 Term 值和索引值
        es[i].Term = r.Term              // Entry 记录的 Term 指定为当前 Leader 节点的任期号
        es[i].Index = li + 1 + uint64(i) // 为日志记录指定 Index
    }
    r.raftLog.append(es...) // 向 raftLog 中追加记录，在前面的 raftLog 小节已经分析过，不再赘述
    // 更新当前节点对应的 Progress，主要是更新 Next 和 Match
    r.prs[r.id].maybeUpdate(r.raftLog.lastIndex())
    r.maybeCommit()    // 尝试提交 Entry 记录
}
```

在 Progress.mayUpdate()方法中，会尝试修改 Match 字段和 Next 字段，用来标识对应节点 Entry 记录复制的情况。Leader 节点除了在向自身 raftLog 中追加记录时（即 appendEntry()方法）会调用该方法，当 Leader 节点收到 Follower 节点的 MsgAppResp 消息（即 MsgApp 消息的响应消息）时，也会调用该方法尝试修改 Follower 节点对应的 Progress 实例。Progress.mayUpdate()方法的具体实现如下：

```
func (pr *Progress) maybeUpdate(n uint64) bool {
    var updated bool
    if pr.Match < n {
        pr.Match = n  // n 之前的成功发送所有 Entry 记录已经写入对应节点的 raftLog 中
```

仅供非商业用途或交流学习使用


```
        updated = true
        // 下面将 Progress.paused 设置为 false，表示 Leader 节点可以继续向对应 Follower 节点
        // 发送 MsgApp 消息（即复制 Entry 记录）
        pr.resume()
    }
    if pr.Next < n+1 {
        pr.Next = n + 1 // 移动 Next 字段，下次要复制的 Entry 记录从 Next 开始
    }
    return updated
}
```

如果该 Entry 记录已经复制到了半数以上的节点中，则在 raft.maybeCommit()方法中会尝试将其提交。除了 appendEntry()方法，在 Leader 节点每次收到 MsgAppResp 消息时也会调用 maybeCommit()方法，maybeCommit()方法的具体实现如下：

```
func (r *raft) maybeCommit() bool {
    // 将集群中所有节点对应的 Progress.Match 字段复制到 mis 切片中
    mis := make(uint64Slice, 0, len(r.prs))
    for id := range r.prs {
        mis = append(mis, r.prs[id].Match)
    }
    sort.Sort(sort.Reverse(mis)) // 对这些 Match 值进行排序
    // raft.quorum()方法返回的值是集群节点的半数+1，这里举个例子：
    // 如果节点数量为 5，r.quorum()-1=2，则可以找到 mis 切片中下标为 2 的节点对应的 Match 值
    // 该值之前的 Entry 记录都是可以提交的，因为节点 0、1、2 三个节点(超半数)已经复制了该记录
    mci := mis[r.quorum()-1]
    return r.raftLog.maybeCommit(mci, r.Term) // 更新 raftLog.committed 字段，完成提交
}
```

到此为止，raft 中与状态切换相关的方法就分析完了。

3. 消息处理

本小节将会按照前面对 Raft 协议的介绍方式，分析节点在不同状态下对不同类型消息的不同处理行为。正如前面分析 newRaft()函数时所看到的，当节点的 raft 实例创建完成之后，当前节点处于 Follower 状态。为了方便介绍，这里假设集群为新建集群，且集群启动后所有节点都会处于 Follower 状态。另外，集群中所有节点的 WAL 日志文件等也都是空的，所以 WAL 日志重放也不会得到任何信息，节点的初始 Term 值也都是 0。另外，我们还假设当前集群开启了

仅供非商业用途或交流学习使用

PreVote 模式以及 CheckQuorum 模式。

➢ MsgHup消息

集群启动一段时间之后，会有一个 Follower 节点的选举计时器超时，此时就会创建 MsgHup 消息（其中 Term 为 0）并调用 raft.Step()方法（参见前面介绍的 raft.tickElection()方法）。raft.Step()方法是 etcd-raft 模块处理各类消息的入口，raft.Step()方法比较长，我们会根据节点的状态及处理的消息类型将其分成不同的代码片段进行介绍。raft.Step()方法主要分为两部分：第一部分是根据 Term 值对消息进行分类处理，第二部分是根据消息的类型进行分类处理。这里先来看一下 MsgHup 消息的处理流程，raft.Step()方法中相关的代码片段如下所示。

```
func (r *raft) Step(m pb.Message) error {
  switch { // 首先根据消息的 Term 值进行分类处理
  case m.Term == 0: // 对本地消息并没有做什么处理，这里介绍的 MsgHup 消息 Term 值为 0，就是本
  // 地消息的一种；后面介绍的 MsgProp 消息和 MsgReadIndex 消息也是本地消息，其 Term 值也是 0
  case m.Term > r.Term: // 暂时不关心该部分处理的代码(略)
  case m.Term < r.Term: // 暂时不关心该部分处理的代码(略)
  }

  switch m.Type { // 根据 Message 的 Type 进行分类处理
  case pb.MsgHup: // 这里针对 MsgHup 类型的消息进行处理
    if r.state != StateLeader { // 只有非 Leader 状态的节点才会处理 MsgHup 消息
      // 获取 raftLog 中已提交但未应用(即 applied~committed)的 Entry 记录，异常处理(略)
      ents, err := r.raftLog.slice(r.raftLog.applied+1,
        r.raftLog.committed+1, noLimit)
      // 检测是否有未应用的 EntryConfChange 记录，如果有就放弃发起选举的机会
      // EntryConfChange 记录的内容后面详细介绍
      if n := numOfPendingConf(ents);
           n != 0 && r.raftLog.committed > r.raftLog.applied {
        return nil
      }

      if r.preVote { // 检测当前集群是否开启了 PreVote 模式，如果开启了，则先切换到
        // 调用 raft.campaign()方法切换当前节点的角色，发起 PreVote
        r.campaign(campaignPreElection)
      } else {
        r.campaign(campaignElection)
      }
    } else { // 如果当前节点已经是 Leader 状态，则仅仅输出一条 Debug 日志
```

仅供非商业用途或交流学习使用


```
            r.logger.Debugf("%x ignoring MsgHup because already leader", r.id)
        }
    case pb.MsgVote, pb.MsgPreVote:  // 对 MsgVote 和 MsgPreVote 消息的处理(略)
    default: // 对于其他类型消息的处理
        r.step(r, m)
    }
    return nil
}
```

在 raft.campaign()方法中，除了完成状态切换，还会向集群中的其他节点发送相应类型的消息，例如，如果当前 Follower 节点要切换成 PreCandidate 状态，则会发送 MsgPreVote 消息。raft.campaign()方法的具体实现如下：

```
func (r *raft) campaign(t CampaignType) {
    // 在该方法的最后，会发送一条消息，这里 term 和 voteMsg 分别用于确定该消息的 Term 值和类型
    var term uint64
    var voteMsg pb.MessageType
    if t == campaignPreElection { // 切换的目标状态是 PreCandidate
        // 将当前节点切换成 PreCandidate 状态，becomePreCandidate()方法的具体实现在前面已
        // 经介绍过了，这里不再赘述
        r.becomePreCandidate()
        voteMsg = pb.MsgPreVote // 确定最后发送的消息是 MsgPreVote 类型
        // 确定最后发送消息的 Term 值，这里只是增加了消息的 Term 值，并未增加 raft.Term 字段的值
        term = r.Term + 1
    } else { // 切换的目标状态是 Candidate
        // 将当前节点切换成 Candidate 状态，需要读者回顾的是，becomeCandidate()方法中会
        // 增加 raft.Term 字段的值，并将当前节点的选票投给自身
        r.becomeCandidate()
        voteMsg = pb.MsgVote // 确定最后发送的消息是 MsgPreVote 类型
        term = r.Term        // 确定最后发送消息的 Term 值
    }
    // 统计当前节点收到的选票，并统计其得票数是否超过半数，这次检测主要是为单节点设置的
    if r.quorum() == r.poll(r.id, voteRespMsgType(voteMsg), true) {
        // 当得到足够的选票时，则将 PreCandidate 状态的节点切换成 Candidate 状态，
        // Candidate 状态的节点则切换成 Leader 状态
        if t == campaignPreElection {
            r.campaign(campaignElection)
        } else {
```

仅供非商业用途或交流学习使用


```
            r.becomeLeader() // becomeLeader()方法在前面已经介绍过，这里不再赘述
        }
        return
    }

    for id := range r.prs { // 状态切换完成之后，当前节点会向集群中所有节点发送指定类型的消息
        if id == r.id { // 跳过当前节点自身
            continue
        }
        var ctx []byte
        // 在进行 Leader 节点转移时，MsgPreVote 或 MsgVote 消息会在 Context 字段中设置该特殊值
        if t == campaignTransfer {
            ctx = []byte(t)
        }
        // 发送指定类型的消息，其中 Index 和 LogTerm 分别是当前节点的 raftLog 中
        // 最后一条消息的 Index 值和 Term 值
        r.send(pb.Message{Term: term, To: id, Type: voteMsg, Index: r.raftLog.lastIndex(),
            LogTerm: r.raftLog.lastTerm(), Context: ctx})
    }
}
```

raft.send()方法会在消息发送之前对不同类型的消息进行合法性检测，然后将待发送的消息追加到 raft.msg 字段中，其具体实现如下：

```
func (r *raft) send(m pb.Message) {
    m.From = r.id   // 设置消息的发送节点 ID，即当前节点 ID
    if m.Type == pb.MsgVote || m.Type == pb.MsgPreVote {
        // 对 MsgVote 和 MsgPreVote 消息的 Term 字段进行检查
        if m.Term == 0 {
            panic(fmt.Sprintf("term should be set when sending %s", m.Type))
        }
    } else {
        if m.Term != 0 { // 对其他类型消息的 Term 字段值进行设置
            panic(fmt.Sprintf("..."))
        }
        // 除了 MsgProp 和 MsgReadIndex 两类消息（这两类消息的 Term 值为 0，即为本地消息）之外，其他
        // 类型消息的 Term 字段值在这里统一设置
        if m.Type != pb.MsgProp && m.Type != pb.MsgReadIndex {
```

仅供非商业用途或交流学习使用


```
            m.Term = r.Term
        }
    }
    r.msgs = append(r.msgs, m) // 将消息添加到 r.msgs 切片中等待发送
}
```

这里简单总结一下，Follower 节点在选举计时器超时的行为：首先它会通过 tickElection() 创建 MsgHup 消息并将其交给 raft.Step()方法进行处理；raft.Step()方法会将当前 Follower 节点切换成 PreCandidate 状态，然后创建 MsgPreVote 类型的消息，最后将该消息追加到 raft.msgs 字段中，等待上层模块将其发送出去。

➢ MsgPreVote消息

当集群中其他节点（此时集群中其他节点都处于 Follower 状态）收到上一小节所述节点发送来的 MsgPreVote（其 Term 字段值为 1）消息之后，经过网络层处理及相关验证之后，最终也会调用 raft.Step()方法进行处理。

下面我们回到 raft.Step()方法中，分析其中关于 MsgPreVote 消息处理的相关代码片段，首先是根据 Term 值进行分类处理的代码片段：

```
switch {
case m.Term == 0: // 本地消息并没有进行什么处理
case m.Term > r.Term:    // 在上述场景中，当收到 MsgPreVote 消息(Term 字段为 1)时，
                         // 集群中的其他 Follower 节点的 Term 值都为 0
    lead := m.From       // 记录消息携带的 Term 值
    // 这里只对 MsgVote 和 MsgPreVote 两种类型消息进行处理
    if m.Type == pb.MsgVote || m.Type == pb.MsgPreVote {

        // 根据消息的 Context 字段判断收到的 MsgPreVote（或 MsgVote）消息是否为 Leader
        // 节点转移场景下产生的，如果是，则强制当前节点参与本次预选（或选举）
        force := bytes.Equal(m.Context, []byte(campaignTransfer))

        // 下面通过一系列条件判断当前节点是否参与此次选举（或预选），其中主要检测集群是否开启
        // 了 CheckQuorum 模式、当前节点是否有已知的 Lead 节点，以及其选举计时器的时间
        inLease := r.checkQuorum && r.lead != None &&
            r.electionElapsed < r.electionTimeout
        if !force && inLease {
            return nil // 当前节点不参与此次选举
        }
```

仅供非商业用途或交流学习使用


```
            lead = None
        }
        switch { // 在这个 switch 中，当前节点会根据消息类型决定是否切换状态
        case m.Type == pb.MsgPreVote: // 收到 MsgPreVote 消息时，不会引起当前节点的状态切换
        // 暂时不关心对其他消息的处理，后面会详细分析(略)
        }
    case m.Term < r.Term: // 暂时不关心该部分处理的代码(略)
    }
```

接下来 raft.Step()方法会根据消息类型进行分类处理，其中会针对 MsgVote 和 MsgPreVote 类型的消息进行单独的处理，相关的代码片段如下：

```
switch m.Type { // 根据消息类型进行分类处理
case pb.MsgHup: // MsgHup 消息的处理，前面已经介绍过了，这里不再赘述
case pb.MsgVote, pb.MsgPreVote:
    // 当前节点在参与预选时，会综合下面几个条件决定是否投票（在 Raft 协议的介绍中也提到过）：
    // 1、当前节点是否已经投过票
    // 2、MsgPreVote 消息发送者的任期号是否更大
    // 3、当前节点投票给了对方节点
    // 4、MsgPreVote 消息发送者的 raftLog 中是否包含当前节点的全部 Entry 记录，isUpToDate()方法
    // 在前面介绍过了，这里不再赘述
    if (r.Vote == None || m.Term > r.Term || r.Vote == m.From) &&
        r.raftLog.isUpToDate(m.Index, m.LogTerm) {
        // 将票投给 MsgPreVote 消息的发送节点
        r.send(pb.Message{To: m.From, Type: voteRespMsgType(m.Type)})
        // 针对 MsgVote 消息的处理(略)
    } else {
        // 不满足上述投赞同票条件时，当前节点会返回拒绝票（即响应消息中的 Reject 字段会设置成 true）
        r.send(pb.Message{To: m.From, Type: voteRespMsgType(m.Type), Reject: true})
    }
default: // 其他类型消息的处理代码暂时省略，后面详细介绍
}
return nil
```

这里简单总结一下 MsgPreVote 消息的处理过程：raft.Step()方法首先检测该 MsgPreVote 消息是否为 Leader 节点迁移时发出的消息及其他合法性检测，决定当前节点是否参与此次选举；之后当前节点会根据自身的状态决定是否将其选票投给 MsgPreVote 消息的发送节点。

仅供非商业用途或交流学习使用

➢ MsgPreVoteResp消息

继续前面的示例，PreCandidate 节点会收到集群中其他节点返回的 MsgPreVoteResp 消息，其中的 Term 字段与 PreCandidate 节点的 Term 值相同。在 raft.Step()方法中没有对 Term 值相等的 MsgPreVoteResp 消息做特殊的处理，而是直接交给了 raft.step 字段指向的函数进行处理。

在前面分析的状态切换方法（become*()方法）中，可以看到 raft.step 字段会根据节点的状态指向不同的消息处理函数，在PreCandidate状态的节点中，该字段指向了 stepCandidate()方法。下面来看一下 raft.stepCandidate()函数对 MsgPreVoteResp 消息的处理逻辑：

```
func stepCandidate(r *raft, m pb.Message) {
    var myVoteRespType pb.MessageType
    if r.state == StatePreCandidate { // 根据当前节点的状态决定其能够处理的选举响应消息的类型
        myVoteRespType = pb.MsgPreVoteResp
    } else {
        myVoteRespType = pb.MsgVoteResp
    }
    switch m.Type {
    // 暂时忽略处理其他类型消息的代码片段
    case myVoteRespType:  // 处理收到的选举响应消息，当前示例中处理的是 MsgPreVoteResp 消息
        gr := r.poll(m.From, m.Type, !m.Reject) // 记录并统计投票结果
        switch r.quorum() { // 得票是否过半数，raft.quorum()方法在前面介绍过了，这里不再赘述
        case gr:
            if r.state == StatePreCandidate {
                // 当 PreCandidate 节点在预选中收到半数以上的选票之后，会发起正式的选举
                r.campaign(campaignElection)
            } else {
                // Candidate 状态节点的处理后面详细介绍(略)
            }
        case len(r.votes) - gr:
            // 赞同票与拒绝票相等时，无法获取半数以上的票数，当前节点切换成 Follower 状态，
            // 等待下一轮的选举(或预选)
            r.becomeFollower(r.Term, None)
        }
    }
}
```

在 raft.poll()方法中会将收到的投票结果记录到 raft.votes 字段中，之后通过统计该字段从而确定该节点的得票数，具体实现如下：

仅供非商业用途或交流学习使用


```
func (r *raft) poll(id uint64, t pb.MessageType, v bool) (granted int) {
    // 根据 MsgPreVoteResp 消息的 Reject 字段输出相应的日志(略)
    if _, ok := r.votes[id]; !ok { // 记录集群中其他节点的投票结果
        r.votes[id] = v
    }
    for _, vv := range r.votes { // 统计投票结果并返回
        if vv {
            granted++
        }
    }
    return granted
}
```

当 PreCandidate 状态节点收到半数以上的投票时，会通过 r.campaign()方法发起正式选举，其中会通过 raft.becomeCandidate()方法将当前节点切换成 Candidate 状态，并向剩余其他节点发送 MsgVote 消息。raft.campaign()方法的具体实现在前面已经介绍过，这里不再展开重复介绍。

➢ MsgVote消息

正如上一小节提到的，PreCandidate 状态节点收到半数以上的投票之后，会发起新一轮的选举，即向集群中的其他节点发送 MsgVote 消息。当集群中其他节点收到 MsgVote 消息之后，也是交由 raft.Step()方法进行处理的，其中根据 Term 值进行分类处理的部分与前面介绍的 MsgPreVote 处理类似，其相关片段如下：

```
switch {
case m.Term == 0: // 本地消息并没有进行什么处理
case m.Term > r.Term:
    lead := m.From    // 记录消息携带的 Term 值
    // 检测当前节点是否要参与此次选举，需要注意的是，这里会将 lead 变量的值设置为 None，
    // 该代码片段在前面已经分析了，这里不再赘述
    switch {
    // MsgPreVote 等其他类型消息的处理(略)
    default:
        r.becomeFollower(m.Term, lead) // 将当前节点切换成 Follower 状态，其中 lead 为 None
    }
case m.Term < r.Term: // 暂时不关心此代码片段(略)
}
```

raft.Step()方法中根据消息类型进行分类处理的代码片段中，除了检测当前节点是否投票及发送 MsgVoteResp 消息，还会重置当前节点的选举超时计时器并更新 raft.Vote 字段，代码比较简单，就不再粘贴出来了。

➢ MsgVoteResp消息

与对 MsgPreVoteResp 消息的处理类似，MsgVoteResp 消息也是由 raft.stepCandidate()方法处理的，其相关的代码实现如下：

```
func stepCandidate(r *raft, m pb.Message) {
  // 根据当前节点的角色决定响应消息的类型(略)
  switch m.Type {
  // 省略处理其他类型消息的代码片段
  case myVoteRespType:  // 处理 MsgVoteResp 消息
    gr := r.poll(m.From, m.Type, !m.Reject) // 记录并统计投票结果
    switch r.quorum() { // 得票是否过半数
    case gr:
      if r.state == StatePreCandidate { // 对 MsgPreVoteResp 消息的处理(略)
      } else {
        // 当前节点切换成为 Leader 状态，其中会重置每个节点对应的 Next 和 Match 两个索引，
        // becomeLeader()方法在前面已经详细分析过了，这里不再赘述
        r.becomeLeader()
        r.bcastAppend()  // 向集群中其他节点广播 MsgApp 消息
      }
    // 选举失败时，将当前节点切换成 Follower 状态，等待下一轮选举(或是预选)
  }
}
```

raft.bcastAppend()方法主要负责向集群中的其他节点发送 MsgApp 消息(或 MsgSnap 消息)，具体实现如下：

```
func (r *raft) bcastAppend() {
  for id := range r.prs {
    if id == r.id { // 过滤当前节点本身，只向集群中其他节点发送消息
      continue
    }
    r.sendAppend(id) // 向指定节点发送消息
  }
}
```

raft.sendAppend()方法主要负责向指定的目标节点发送 MsgApp 消息（或 MsgSnap 消息），在消息发送之前会检测当前节点的状态，然后查找待发送的 Entry 记录并封装成 MsgApp 消息，之后根据对应节点的 Progress.State 值决定发送消息之后的操作。下面先描述一下 Progress.State 各值的含义：

- ProgressStateSnapshot 状态表示 Leader 节点正在向目标节点发送快照数据。
- ProgressStateProbe 状态表示 Leader 节点一次不能向目标节点发送多条消息，只能待一条消息被响应之后，才能发送下一条消息。当刚刚复制完快照数据、上次 MsgApp 消息被拒绝（或是发送失败）或是 Leader 节点初始化时，都会导致目标节点的 Progress 切换到该状态。
- ProgressStateReplicate 状态表示正常的 Entry 记录复制状态，Leader 节点向目标节点发送完消息之后，无须等待响应，即可开始后续消息的发送。

Progress.State 字段的状态转换如图 3-11 所示。

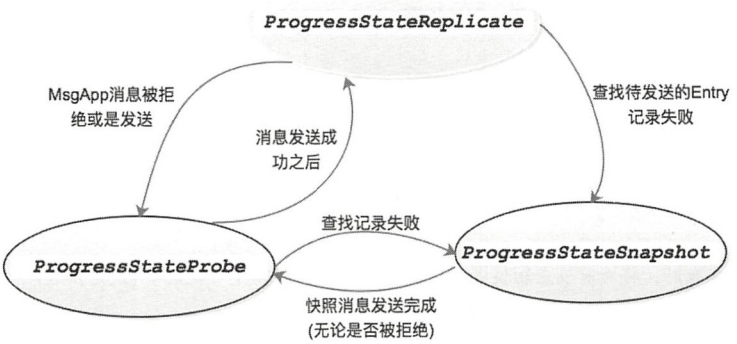


图 3-11

下面我们继续来看一下 raft.sendAppend()方法的具体实现。

```
func (r *raft) sendAppend(to uint64) {
  pr := r.prs[to]
  if pr.IsPaused() { // 检测当前节点是否可以向目标节点发送消息
    return
  }
  m := pb.Message{} // 创建待发送的消息
  m.To = to // 设置目标节点的 ID

  // 根据当前 Leader 节点记录的 Next 查找发往指定节点的 Entry 记录(ents)及 Next 索引对
```

```
    // 应的记录的 Term 值(term)
    term, errt := r.raftLog.term(pr.Next - 1)
    ents, erre := r.raftLog.entries(pr.Next, r.maxMsgSize)

    // 上述两次 raftLog 查找出现异常时，就会形成 MsgSnap 消息，将快照数据发送到指定节点，
    // 后面详细介绍 MsgSnap 消息的发送，这里重点看一下 MsgApp 消息发送
    if errt != nil || erre != nil {
        // 上述两次 raftLog 查找出现异常时，就会形成 MsgSnap 消息，将快照数据发送到指定节点，
        // 这部分代码暂时省略，后面详细介绍 MsgSnap 消息的发送，这里重点看一下 MsgApp 消息发送
    } else {
        m.Type = pb.MsgApp // 设置消息类型
        m.Index = pr.Next - 1 // 设置 MsgApp 消息的 Index 字段
        m.LogTerm = term // 设置 MsgApp 消息的 LogTerm 字段
        m.Entries = ents // 设置消息携带的 Entry 记录集合
        // 设置消息的 Commit 字段，即当前节点的 raftLog 中最后一条已提交的记录索引值
        m.Commit = r.raftLog.committed
        if n := len(m.Entries); n != 0 {
            switch pr.State { // 根据目标节点对应的 Progress.State 状态决定其发送消息后的行为
            case ProgressStateReplicate: // ProgressStateReplicate 状态下的处理
                last := m.Entries[n-1].Index
                pr.optimisticUpdate(last) // 更新目标节点对应的 Next 值(这里不会更新 Match 值)
                pr.ins.add(last)          // 记录已发送但是未收到响应的消息
            case ProgressStateProbe: // ProgressStateProbe 状态下的处理
                // 消息发送后，就将 Progress.Paused 字段设置成 true，暂停后续消息的发送
                pr.pause()
            default:
                r.logger.Panicf("%x is sending append in unhandled state %s", r.id, pr.State)
            }
        }
    }
    // 发送前面创建的 MsgApp 消息，raft.send()方法会设置 MsgApp 消息的 Term 字段值，
    // 并将其追加到 raft.msgs 中等待发送。raft.send()方法前面介绍过了，这里不再展开介绍
    r.send(m)
}
```

在 Progress.IsPaused()方法中会检测当前节点是否能够向对应的节点发送消息，具体实现如下：

```
func (pr *Progress) IsPaused() bool {
  switch pr.State {
  case ProgressStateProbe: // ProgressStateProbe 状态时检测 Paused 字段
    return pr.Paused
  case ProgressStateReplicate: // ProgressStateReplicate 状态时检测已发送未响应的消息个数
    return pr.ins.full()
  case ProgressStateSnapshot: // ProgressStateSnapshot 状态时始终可以发送消息
    return true
  default: panic("unexpected state")
  }
}
```

这里简单总结一下 MsgVoteResp 消息的处理过程：raft.Step()方法首先检测当前节点是否收到了半数以上的选票，如果是，则将当前节点切换成 Leader 状态；之后向集群中其他节点发送消息。

➢ MsgApp消息

通过对前面的示例描述可知，在 MsgVote 消息的处理过程中，集群中其他节点已经切换成了 Follower 状态，并且它们自身记录的 Term 值与该任期中 Leader 节点维护的 Term 值相同。当它们收到当前 Leader 节点发来的 MsgApp 消息时，也是交由 raft.Step()方法处理的。下面分析其中相关的代码片段，如下：

```
func (r *raft) Step(m pb.Message) error {
  switch {
  // 这里只处理消息 Term 字段为 0，以及消息 Term 字段与当前节点不相等的场景(略)
  }

  switch m.Type {
  // 处理 MsgHup、MsgVote、MsgPreVote 消息的代码片段省略
  default:
    r.step(r, m) // 当前节点是 Follower 状态，raft.step 字段指向 stepFollower()函数
  }
  return nil
}
```

继续来看一下 raft.stepFollower()方法中对 MsgApp 消息的处理，相关代码片段如下：

```
func stepFollower(r *raft, m pb.Message) {
```

```
    switch m.Type {
    // 这里重点来看一下 MsgApp 消息的处理，其他类型消息的处理过程暂时忽略
    case pb.MsgApp:
      r.electionElapsed = 0 // 重置选举计算器，防止当前 Follower 发起新一轮选举
      r.lead = m.From       // 设置 raft.Lead 记录，保存当前集群的 Leader 节点 ID
      // 将 MsgApp 消息中携带的 Entry 记录追加到 raftLog 中， 并且向 Leader 节点发送 MsgAppResp
      // 消息，响应此次 MsgApp 消息
      r.handleAppendEntries(m)
  }
```

raft.handleAppendEntries()方法首先会检测 MsgApp 消息中携带的 Entry 记录是否合法，然后将这些Entry记录追加到raftLog中，最后创建相应的MsgAppResp消息。handleAppendEntries()方法的具体实现如下：

```
  func (r *raft) handleAppendEntries(m pb.Message) {
    // m.Index 表示 leader 发送给 follower 的上一条日志的索引位置，
    // 如果 Follower 节点在 Index 位置的 Entry 记录已经提交过了，
    // 则不能进行追加操作，在前面在 Raft 协议时提到过，已提交的记录不能被覆盖，
    // 所以 Follower 节点会将其 committed 位置通过 MsgAppResp 消息(Index 字段)通知 Leader 节点
    if m.Index < r.raftLog.committed {
      r.send(pb.Message{To: m.From, Type: pb.MsgAppResp, Index: r.raftLog.committed})
      return
    }
    // 尝试将消息携带的 Entry 记录追加到 raftLog 中
    if mlastIndex, ok := r.raftLog.maybeAppend(m.Index, m.LogTerm, m.Commit,
m.Entries...);
    ok {
      // 如果追加成功，则将最后一条记录的索引值通过 MsgAppResp 消息返回给 Leader 节点，
      // 这样 Leader 节点就可以根据此值更新其对应的 Next 和 Match 值
      r.send(pb.Message{To: m.From, Type: pb.MsgAppResp, Index: mlastIndex})
    } else {
      // 如果追加记录失败，则将失败信息返回给 Leader 节点(即 MsgAppResp 消息的 Reject 字段为 true)，
      // 同时返回的还有一些提示信息(RejectHint 字段保存了当前节点 raftLog 中最后一条记录的索引)
      r.send(pb.Message{To: m.From, Type: pb.MsgAppResp, Index: m.Index,
        Reject: true, RejectHint: r.raftLog.lastIndex()})
    }
  }
```

仅供非商业用途或交流学习使用

raftLog.maybeAppend()方法和 raft.send()方法的具体实现在前面已经分析过了，这里不再展开详述。

➢ MsgAppResp消息

接下来我们回到 Leader 节点，当其收到集群中其他 Follower 节点发送的 MsgAppResp 响应消息之后，也是交由 raft.Step()方法进行处理的，其中会调用 raft.step 字段指向的 StepLeader()函数进行处理的。下面开始分析 raft.StepLeader()方法中处理 MsgAppResp 消息的代码片段：

```
func stepLeader(r *raft, m pb.Message) {
    // 下面主要注册 MsgBeat、MsgCheckQuorum、MsgProp、MsgReadIndex 消息，这些消息无须处理
    // 消息的 From 字段和消息发送者对应的 Progress 实例可以直接进行处理，具体处理代码省略
    switch m.Type {
    case pb.MsgBeat:
    case pb.MsgCheckQuorum:
    case pb.MsgProp:
    case pb.MsgReadIndex:
    }
    // 根据消息的 From 字段获取对应的 Progress 实例，为后面的消息处理做准备
    pr, prOk := r.prs[m.From]
    if !prOk { // 没有操作到对应的 Progress 实例(该节点可能已从集群中删除)时，直接返回
        return
    }
    switch m.Type {
    // 这里重点来看一下 MsgAppResp 消息的处理，处理其他类型消息的代码片段暂时忽略，后面遇到时详细分析
    case pb.MsgAppResp:
        // 更新对应 Progress 实例的 RecentActive 字段，从 Leader 节点的角度来看，MsgAppResp 消息
        // 的发送节点还是存活的
        pr.RecentActive = true
        if m.Reject { // MsgApp 消息被拒绝
            // 通过 MsgAppResp 消息携带的信息及对应的 Progress 状态，重新设置其 Next 值
            if pr.maybeDecrTo(m.Index, m.RejectHint) {
                // 如果对应的 Progress 处于 ProgressStateReplicate 状态，则切换成
                // ProgressStateProbe 状态，试探 Follower 的匹配位置(这里的试探是指发送一条消息并
                // 等待其相应之后，再发送后续的消息)
                if pr.State == ProgressStateReplicate {
                    pr.becomeProbe()
                }
                // 再次向对应 Follower 节点发送 MsgApp 消息，读者可以回忆前面的介绍，在 sendAppend()
```

仅供非商业用途或交流学习使用

```
            // 方法中会将对应的 Progress.pause 字段设置为 true，从而暂停后续消息的发送，从而实现
            // 前面说的“试探”的效果
            r.sendAppend(m.From)
        }
    } else { // 之前发送的 MsgApp 消息已经被对应的 Follower 节点接收(Entry 记录被成功追加)
        oldPaused := pr.IsPaused()
        // MsgAppResp 消息的 Index 字段是对应 Follower 节点 raftLog 中最后一条 Entry 记录的索
        // 引，这里会根据该值更新其对应 Progress 实例的 Match 和 Next，Progress.maybeUpdate()
        // 方法在前面已经介绍过了，这里不再赘述
        if pr.maybeUpdate(m.Index) {
            switch {
            case pr.State == ProgressStateProbe:
                // 一旦 MsgApp 被 Follower 节点接收，则表示已经找到其正确的 Next 和 Match，
                // 不必再进行“试探”，这里将对应的 Progress.state 切换成 ProgressStateReplicate
                pr.becomeReplicate()
            case pr.State == ProgressStateSnapshot && pr.needSnapshotAbort():
                // 之前由于某些原因，Leader 节点通过发送快照的方式恢复 Follower 节点，但在发送
                // MsgSnap 消息的过程中，Follower 节点恢复，并正常接收了 Leader 节点的 MsgApp 消
                // 息，此时会丢弃 MsgSnap 消息，并开始“试探”该 Follower 节点正确的 Match 和 Next 值
                pr.becomeProbe()
            case pr.State == ProgressStateReplicate:
                // 之前向某个 Follower 节点发送 MsgApp 消息时，会将其相关信息保存到对应的
                // Progress.ins 中，在这里收到相应的 MsgAppResp 响应之后，会将其从 ins 中删除，
                // 这样可以实现了限流的效果，避免网络出现延迟时，继续发送消息，从而导致网络更加拥堵
                pr.ins.freeTo(m.Index)
            }

            // 收到一个 Follower 节点的 MsgAppResp 消息之后，除了修改相应的 Match 和 Next 值，
            // 还会尝试更新 raftLog.committed，因为有些 Entry 记录可能在此次复制中被保存到了
            // 半数以上的节点中，raft.maybeCommit()方法在前面已经分析过了，这里不再赘述
            if r.maybeCommit() {
                // 向所有节点发送 MsgApp 消息，注意，此次 MsgApp 消息的 Commit 字段与
                // 上次 MsgApp 消息已经不同，raft.bcastAppend()方法不再赘述
                r.bcastAppend()
            } else if oldPaused { // 之前是 pause 状态，现在可以任性地发消息了
                // 之前 Leader 节点暂停向该 Follower 节点发送消息，收到 MsgAppResp 消息后，在上述
                // 代码中已经重置了相应状态，所以可以继续发送 MsgApp 消息
                r.sendAppend(m.From)
```

仅供非商业用途或交流学习使用

```
                }
                // Leader 节点迁移的相关代码(略)
            }
        }
    }
}
```

stepLeader()方法关于 MsgAppResp 消息处理的代码片段中，大部分的调用在前面都详细介绍过了，这里介绍一下两个第一次遇到的方法。首先是 Progress.maybeDecrTo()方法，它会根据对应 Progress 的状态和 MsgAppResp 消息携带的提示信息，完成 Progress.Next 的更新，具体实现如下：

```
// maybeDecrTo()方法的两个参数都是 MsgAppResp 消息携带的信息：
// reject 是被拒绝 MsgApp 消息的 Index 字段值，
// last 是被拒绝 MsgAppResp 消息的 RejectHint 字段值（即对应 Follower 节点 raftLog 中最后一条
// Entry 记录的索引）
func (pr *Progress) maybeDecrTo(rejected, last uint64) bool {
    if pr.State == ProgressStateReplicate {
        if rejected <= pr.Match { // 出现过时的 MsgAppResp 消息，直接忽略
            return false
        }
        // 根据前面对 MsgApp 消息发送过程的分析，处于 ProgressStateReplicate 状态时，发送 MsgApp
        // 消息的同时会直接调用 Progress.optimisticUpdate()方法增加 Next，这就使得 Next 可能会
        // 比 Match 大很多，这里回退 Next 至 Match 位置，并在后面重新发送 MsgApp 消息进行尝试
        pr.Next = pr.Match + 1
        return true
    }

    if pr.Next-1 != rejected { // 出现过时的 MsgAppResp 消息，直接忽略
        return false
    }
    // 根据 MsgAppResp 携带的信息重置 Next
    if pr.Next = min(rejected, last+1); pr.Next < 1 {
        pr.Next = 1 // 将 Next 重置为 1
    }
    pr.resume() // Next 重置完成，恢复消息发送，并在后面重新发送 MsgApp 消息
    return true
}
```



仅供非商业用途或交流学习使用

另一个需要介绍的是 inflights 结构体，Progress.ins 字段就指向了一个 inflights 实例，inflights 的主要功能是记录当前节点已发出但未收到响应的 MsgApp 消息。inflights 中各个字段的含义如下。

- buffer([]uint64 类型): 用来记录 MsgApp 消息相关信息的数组，其中记录的是 MsgApp 消息中最后一条 Entry 记录的索引值。
- start (int 类型): inflights.buffer 数组被当作一个环形数组使用，start 字段中记录 buffer 中第一条 MsgApp 消息的下标。
- count (int 类型): 当前 inflights 实例中记录的 MsgApp 消息个数。
- size (int 类型): 当前 inflights 实例中能够记录的 MsgApp 消息个数的上限。

在前面介绍 MsgApp 消息的发送过程时，提到过 inflights.add()方法是用来记录发送出去的 MsgApp 消息，其具体实现如下：

```
func (in *inflights) add(inflight uint64) {
  if in.full() { // 检测当前 buffer 数组是否已经被填充满了
    panic("cannot add into a full inflights")
  }
  next := in.start + in.count // 获取新增消息的下标
  size := in.size
  if next >= size { // 环形队列
    next -= size
  }
  // 初始化时的 buffer 数组较短，随着使用会不断进行扩容(两倍)，但其扩容的上限为 size
  if next >= len(in.buffer) {
    in.growBuf() // 扩容的逻辑比较简单，这里就不再贴出代码了，感兴趣的读者可以参考其代码进行学习
  }
  in.buffer[next] = inflight // 在 next 位置记录消息中最后一条 Entry 记录的索引值
  in.count++                 // 递增 count 字段
}
```

通过前面分析的内容可知，当 Leader 节点收到 MsgAppResp 消息时，会通过 inflights.freeTo() 方法将指定消息及其之前的消息全部清空，释放 inflights 空间，让后面的消息继续发送。inflights.freeTo()方法的具体实现如下：

```
func (in *inflights) freeTo(to uint64) {
  // 边界检测，检测当前 inflights 是否为空，以及参数 to 是否有效(略)
```



仅供非商业用途或交流学习使用

```
    i, idx := 0, in.start
    for i = 0; i < in.count; i++ { // 从 start 开始遍历 buffer
        if to < in.buffer[idx] { // 查找第一个大于指定索引值的位置
            break
        }
        size := in.size
        // 因为是环形队列，如果 idx 越界，则从 0 开始继续遍历
        if idx++; idx >= size {
            idx -= size
        }
    }
    in.count -= i  // i 记录了此次释放的消息个数
    in.start = idx // 从 start~idx 的所有消息都被释放（注意，是环形队列）
    if in.count == 0 { // inflights 中全部消息都被清空了，则重置 start
        in.start = 0
    }
}
```

inflights 还提供了一些其他方法，例如，inflights.full()方法用来判断当前 inflights 实例是否被填充满了，inflights.freeFirstOne()方法只释放其中记录的第一条 MsgApp 消息。这些方法比较简单，不再详细分析，感兴趣的读者可以参考其源码进行学习。

➢ MsgBeat消息和MsgCheckQuorum消息

在上述示例中，Leader 节点除了向集群中其他 Follower 节点发送 MsgApp 消息，还会向这些 Follower 节点发送 MsgBeat 消息。MsgBeat 消息的主要作用是探活，当 Follower 节点收到 MsgBeat 消息时会重置其选举计时器，从而防止 Follower 节点发起新一轮选举。

在前面介绍 becomeLeader()方法时提到，当节点切换成 Leader 状态时，会将 raft.tick 字段设置成 raft.tickHeartbeat()方法。下面先来看一下 tickHeartbeat()方法的具体实现：

```
func (r *raft) tickHeartbeat() {
    r.heartbeatElapsed++ // 递增心跳计时器
    r.electionElapsed++ // 递增选举计时器

    if r.electionElapsed >= r.electionTimeout {
        r.electionElapsed = 0 // 重置选举计时器，Leader 节点不会主动发起选举
        if r.checkQuorum { // 进行多数检测
            // 当选举计时器超过 electionTimeout 时，会触发一次 checkQuorum 操作
```



仅供非商业用途或交流学习使用

```
            // 该操作并不会发送网络消息(虽然这里会通过 raft.Step()方法处理消息),
            // 它只是检测当前节点是否与集群中的大部分节点连通
            r.Step(pb.Message{From: r.id, Type: pb.MsgCheckQuorum})
        }
        // 选举计时器处于 electionTimeout~randomizedElectionTimeout 时间段时,
        // 不能进行 Leader 节点的转移
        if r.state == StateLeader && r.leadTransferee != None {
            r.abortLeaderTransfer() // 清空 raft.leadTransferee 字段, 放弃转移
        }
    }

    if r.state != StateLeader { // 检测当前节点是否为 Leader 节点
        return
    }
    if r.heartbeatElapsed >= r.heartbeatTimeout { // 心跳计时器超时
        r.heartbeatElapsed = 0 // 重置心跳计时器
        r.Step(pb.Message{From: r.id, Type: pb.MsgBeat}) // 发送 MsgBeat 消息
    }
}
```

通过 raft.tickHeartbeat()方法的代码可知，MsgCheckQuorum 消息和 MsgBeat 消息的 Term 字段都为 0，所以它们也属于本地消息的一种。raft.Step()方法并没有对 MsgCheckQuorum 消息和 MsgBeat 消息进行特殊的处理，其主要处理是在 raft.stepLeader()方法中，相关代码片段如下：

```
func stepLeader(r *raft, m pb.Message) {
    // 省略处理其他类型消息的代码片段
    switch m.Type {
    case pb.MsgBeat:        // MsgBeat 消息的处理
        r.bcastHeartbeat() // 向所有节点发送信心跳
        return
    case pb.MsgCheckQuorum: // MsgCheckQuorum 消息的处理
        // 用于检测当前 Leader 节点是否与集群中大部分节点连通, 如果不连通, 则切换成 Follower 状态
        if !r.checkQuorumActive() {
            r.becomeFollower(r.Term, None) // raft.becomeFollower()方法在前面介绍过, 这里
                                           // 不再赘述
        }
        return
    }
}
```



仅供非商业用途或交流学习使用

其中，raft.checkQuorumActive()方法会检测当前集群中与当前 Leader 节点连通的节点个数是否超过了半数，具体实现如下：

```
func (r *raft) checkQuorumActive() bool {
  var act int
  for id := range r.prs { // 遍历集群中全部节点对应的 Progress 实例
    if id == r.id { // 当前节点自身对应的 Progress 实例
      act++
      continue
    }
    // 检测集群中其他节点的 Progress.RecentActive，读者可以回忆前面介绍的 MsgAppResp 消息处
    // 理过程，当 Leader 节点接收到 Follower 节点返回的 MsgAppResp 消息时，会将其对应的
    // Progress.RecentActive 设置为 true。后面将要介绍的 MsgBeatResp 消息处理中，也有类似的
    // 操作，所以这里可以直接通过 RecentActive 字段判断对应 Follower 节点的连通性
    if r.prs[id].RecentActive {
      act++
    }
    r.prs[id].RecentActive = false // 重置 RecentActive 字段
  }
  return act >= r.quorum() // 检测与当前节点连通的节点个数是否超过半数
}
```

了解了 MsgCheckQuorum 消息的处理过程之后，我们再来了解一下 raft.bcastHeartbeat()方法，它通过调用 raft.bcastHeartbeatWithCtx()方法向集群中其他节点发送 MsgHeartbeat 消息。bcastHeartbeatWithCtx()方法的具体实现如下：

```
func (r *raft) bcastHeartbeatWithCtx(ctx []byte) {
  for id := range r.prs {
    if id == r.id { // 过滤当前节点自身
      continue
    }
    r.sendHeartbeat(id, ctx) // 向指定的节点发送 MsgBeat 消息
  }
}

// 下面是 raft.sendHeartbeat()方法的具体实现
func (r *raft) sendHeartbeat(to uint64, ctx []byte) {
```



```
    // 注意 MsgHeartbeat 消息中 Commit 字段的设置，这主要是因为在发送该 MsgHeartbeat 消息时，
    // Follower 节点并不一定已经收到了全部已提交的 Entry 记录
    commit := min(r.prs[to].Match, r.raftLog.committed)
    m := pb.Message{
        To:      to,
        Type:    pb.MsgHeartbeat,
        Commit:  commit,
        Context: ctx,
    }
    r.send(m) // raft.send()方法在前面介绍过了，这里不再赘述
}
```

➢ MsgHeartbeat消息和MsgHeartbeatResp消息

当集群中的 Follower 节点收到 Leader 节点发来的 MsgHeartbeat 消息之后，也是通过 raft.Step()方法调用 raft.stepFollower()方法进行处理的，处理完成后会向 Leader 节点返回相应的 MsgHearbeatResp 消息作为响应。raft.stepFollower()方法中的相关代码片段如下：

```
func stepFollower(r *raft, m pb.Message) {
    switch m.Type {
    // 这里重点关注 MsgHeartbeat 消息的处理，其他类型消息相关的代码片段这里不再粘贴
    case pb.MsgHeartbeat:     // MsgHeartbeat 消息的处理
        r.electionElapsed = 0 // 重置选举计时器，防止当前 Follower 节点发起新一轮选举
        r.lead = m.From       // 设置 Leader 节点的 id
        // 其中会修改 raft.committed 字段的值（放心，在 Leader 发送消息时，
        // 已经确定了当前 Follower 节点有 committed 之前的全部日志记录），
        // 然后发送 MsgHeartbeatResp 类型消息，响应此次心跳
        r.handleHeartbeat(m)
    }
}

// 下面是 handleHeartbeat()方法的具体实现
func (r *raft) handleHeartbeat(m pb.Message) {
    // 根据 MsgHeartbeat 消息的 Commit 字段，更新 raftLog 中记录的已提交位置，注意，在 Leader 节点
    // 发送 MsgHeartbeat 消息时，已经确定了当前 Follower 节点中 raftLog.committed 字段的合适位置
    r.raftLog.commitTo(m.Commit)
    // 发送 MsgHeartbeatResp 消息，响应心跳，raft.send()方法在前面已经介绍过了，这里不再赘述
    r.send(pb.Message{To: m.From, Type: pb.MsgHeartbeatResp, Context: m.Context})
}
```



下面再回到 Leader 节点中，分析其 MsgHearbeatResp 消息的相关处理，stepLeader()方法中的相关代码片段如下：

```
func stepLeader(r *raft, m pb.Message) {
    // 根据 MsgHeartbeatResp 消息的 From 字段查找 Follower 节点对应的 Progress 实例
    pr, prOk := r.prs[m.From]
    if !prOk {
        return
    }
    switch m.Type { // 这里主要介绍 MsgHeartbeatResp 消息的处理，其他消息的代码片段省略
    case pb.MsgHeartbeatResp:
        pr.RecentActive = true // 更新对应 Progress.RecentActive，表示 Follow 节点与自己连通
        pr.resume()    // 重置 Progress.Paused 字段，表示可以继续向 Follower 节点发送消息

        if pr.State == ProgressStateReplicate && pr.ins.full() {
            pr.ins.freeFirstOne() // 释放 inflights 中第一个消息，这样就可以开始后续消息的发送
        }
        // 当 Leader 节点收到 Follower 节点的 MsgHeartbeat 消息之后，会比较对应的 Match 值与 Leader
        // 节点的 raftLog，从而判断 Follower 节点是否已拥有了全部的 Entry 记录
        if pr.Match < r.raftLog.lastIndex() {
            r.sendAppend(m.From) // 通过向指定节点发送 MsgApp 消息完成 Entry 记录的复制
        }
        // 后面还有关于只读请求相关的 MsgHeartbeatResp 消息的处理，暂时省略，后面介绍 MsgReadIndex
        // 消息时再详细分析
    }
}
```

➢ MsgProp消息

在 etcd-raft 模块中，客户端发往到集群的写请求是通过 MsgProp 消息表示的。在前面介绍 Raft 协议时提到，Raft 集群中只有 Leader 节点能够响应客户端的写入请求。Leader 节点对于 MsgProp 消息的主要处理是在 raft.stepLeader()方法中实现的，相关的代码片段如下：

```
func stepLeader(r *raft, m pb.Message) {
    switch m.Type {
    case pb.MsgProp: // 这里重点介绍 MsgProp 消息相关代码片段，其他类型消息的处理片段暂时省略
        // 检测 MsgProp 消息是否携带了 Entry 记录，如果未携带，则输出异常日志并终止程序(略)
        // 检测当前节点是否被移出集群，如果当前节点以 Leader 状态被移出集群，则不再处理 MsgProp 消息(略)
        // 检测当前是否正在进行 Leader 节点的转移，不再处理 MsgProp 消息(略)
```



```
        for i, e := range m.Entries { // 遍历 MsgProp 消息携带的全部 Entry 记录
            // 如果存在 EntryConfChange 类型的 Entry 记录，则将 raft.pendingConf 设置为 true
            if e.Type == pb.EntryConfChange {
                if r.pendingConf { // 如果存在多条 EntryConfChange 类型的记录，则只保留第一条
                    m.Entries[i] = pb.Entry{Type: pb.EntryNormal}
                }
                r.pendingConf = true
            }
        }
        r.appendEntry(m.Entries...) // 将上述 Entry 记录追加到当前节点的 raftLog 中
        // 通过 MsgApp 消息向集群中其他节点复制 Entry 记录，bcastAppend()方法在前面介绍过了，这里
        // 不再赘述
        r.bcastAppend()
        return
    }
}
```

当集群中的 Candidate 节点收到客户端发来的 MsgProp 消息时，会直接忽略该消息。当 Follower 节点收到 MsgProp 消息时，会将该 MsgProp 消息转发给当前集群的 Leader 节点，stepFollower()方法中相关代码片段如下：

```
func stepFollower(r *raft, m pb.Message) {
    switch m.Type {    // 这里重点介绍 MsgProp 消息相关代码片段，其他类型消息的处理片段暂时省略
    case pb.MsgProp:
        if r.lead == None { // 当前集群中没有 Leader 节点，则忽略该 MsgProp 消息
            return
        }
        m.To = r.lead // 将消息的 To 字段设置为当前 Leader 节点的 id
        r.send(m) // 将 MsgProp 消息发送到当前的 Leader 节点，send()方法前面介绍过，这里不再赘述
    }
}
```

➢ MsgReadIndex消息和MsgReadIndexResp消息

在前面介绍 Raft 协议时提到，客户端的读请求需要读到集群中最新的、已提交的数据（即 linearizability 语义），而不能读到老数据。在现实场景中也经常会遇到读多写少的情况，如果每次读请求都涉及多个节点的磁盘操作，则性能必然较差。Leader 节点保存了整个集群中最新的数据，如果只读请求只访问 Leader 节点，则 Leader 节点可以直接将结果返回给客户端，但是在



网络分区的场景下，一个旧的 Leader 节点就可能返回旧数据。

为此，etcd-raft 模块使用 MsgReadIndex 消息来解决上述问题：当 Leader 节点收到客户端的只读请求时，会将当前请求的编号记录下来，在返回数据给客户端之前，Leader 节点需要先确定自己是否依然是当前集群的 Leader 节点（通过心跳的方式），在确定其依然是 Leader 节点之后，就可以说明该节点可以响应该请求，只需要等待当前 Leader 节点的提交位置（即 raftLog.committed）到达或是超过只读请求的编号即可向客户端返回响应。

在 etcd-raft 模块中，客户端发往集群的只读请求使用 MsgReadIndex 消息表示，其中只读请求有两种模式，分别是 ReadOnlySafe 和 ReadOnlyLeaseBased。Leader 节点对于 MsgReadIndex 消息的主要处理也是在 raft.stepLeader()方法中实现的，其相关代码片段如下：

```
func stepLeader(r *raft, m pb.Message) {
  switch m.Type {
  case pb.MsgReadIndex: // 重点介绍 MsgReadIndex 消息相关代码片段，其他类型消息的代码片段省略
    if r.quorum() > 1 { // 集群场景
      // Leader 节点检测自身在当前任期中是否已提交过 Entry 记录，如果没有，
      // 则无法进行读取操作(可以回顾前面对 Raft 协议的介绍)，直接返回(略)
      switch r.readOnly.option {
      // 两种处理读请求的模式，这里先来看一下 ReadOnlySafe 模式下的相关代码片段，
      // ReadOnlyLeaseBased 模式下的相关代码在后面单独介绍
      case ReadOnlySafe:
        // 记录当前节点的 raftLog.committed 字段值，即已提交的位置
        r.readOnly.addRequest(r.raftLog.committed, m)
        r.bcastHeartbeatWithCtx(m.Entries[0].Data) // 发送心跳
      }
    } else { // 单节点的情况
      r.readStates = append(r.readStates,
              ReadState{Index: r.raftLog.committed, RequestCtx: m.Entries[0].Data})
    }
    return
  }
}
```

这里涉及 raft.readOnly 字段（*readOnly 类型），该字段的主要作用就是批量处理只读请求。结构体 readOnly 中的字段含义如下：

- option（ReadOnlyOption 类型）：当前只读请求的处理模式，正如前面介绍的那样，有 ReadOnlySafe 和 ReadOnlyLeaseBased 两种模式。



- pendingReadIndex（map[string]*readIndexStatus 类型）：在 etcd 服务端收到 MsgReadIndex 消息时，会为期创建一个消息 ID（该 ID 是唯一的），并作为 MsgReadIndex 消息的第一条 Entry 记录。在 pendingReadIndex 中记录了消息 ID 与对应请求的 readIndexStatus 实例的映射。readIndexStatus 中的字段如下。
  - req（pb.Message 类型）：记录了对应的 MsgReadIndex 请求。
  - index（uint64 类型）：该 MsgReadIndex 请求到达时，对应的已提交位置（即 raftLog.committed）。
  - acks(map[uint64]struct{}类型):记录了该 MsgReadIndex 相关的 MsgHeartbeatrResp 响应信息。
- readIndexQueue（[]string 类型）：记录了 MsgReadIndex 请求对应的消息 ID。

下面回到 raft.stepLeader()方法中，在 ReadOnlySafe 模式下首先会通过 ReadOnly.addRequest() 方法将已提交位置（raftLog.committed）及 MsgReadIndex 消息的相关信息记录到 raft.readOnly 中，ReadOnly.addRequest()方法的具体实现如下：

```
func (ro *readOnly) addRequest(index uint64, m pb.Message) {
  ctx := string(m.Entries[0].Data) // 在 ReadIndex 消息的第一个记录中，记录了消息 ID
  // 检测是否存在 ID 相同的 MsgReadIndex，如果存在，则不再记录该 MsgReadIndex 请求
  // 创建 MsgReadIndex 对应的 readIndexStatus 实例，并记录到 pendingReadIndex 中
  ro.pendingReadIndex[ctx] = &readIndexStatus{index: index, req:
       m, acks: make(map[uint64]struct{})}
  ro.readIndexQueue = append(ro.readIndexQueue, ctx) // 记录消息 ID
}
```

之后调用 raft.bcastHeartbeatWithCtx()方法，向集群中其他节点发送 MsgHeartbeat 消息，此次发送的 MsgHeartbeat 消息中会将上述生成的 ID 作为 Context 字段的值。raft.bcastHeartbeatWithCtx()方法在前面已经介绍过了，这里不再赘述。

前面介绍 MsgHeartbeat 消息处理时分析过，Follower 节点通过 raft.stepFollower()方法完成对 MsgHeartbeat 消息的处理，这里读者回顾的是，在创建 MsgHeartbeatResp 消息时，会将其 Context 字段值设置为 MsgHeartbeat 消息携带的 Context 值。

接下来回到 raft.stepLeader()方法对 MsgHeartbeatResp 消息的处理，在前面对 MsgHeartbeatResp 消息的介绍中，已经介绍了 Leader 节点对普通 MsgHeartbeatResp 消息的处理，这里重点来看一下其中对携带 Context 字段的 MsgHeartbeatResp 消息的处理，相关代码片段如下：

```
func stepLeader(r *raft, m pb.Message) {
```



```
    switch m.Type {
    case pb.MsgHeartbeatResp:
        // 前面对所有 MsgHeartbeatResp 消息的处理在前面已经详细介绍过了，这里不再赘述
        // 下面只处理 ReadOnlySafe 模式下，只读请求相关的 MsgHeartbeatResp 消息
        if r.readOnly.option != ReadOnlySafe || len(m.Context) == 0 {
            return
        }
        // 统计目前为止响应上述携带消息 ID 的 MsgHeartbeat 消息的节点个数
        ackCount := r.readOnly.recvAck(m)
        if ackCount < r.quorum() {
            return
        }

        // 响应节点超过半数之后，会清空 readOnly 中指定消息 ID 及其之前的所有相关记录，具体实现下面
        // 详细描述
        rss := r.readOnly.advance(m)
        for _, rs := range rss {
            req := rs.req
            if req.From == None || req.From == r.id {
                // 根据 MsgReadIndex 消息的 From 字段，判断该 MsgReadIndex 消息是否为 Follower
                // 节点转发到 Leader 节点的消息
                // 如果是客户端直接发送到 Leader 节点的消息，则将 MsgReadIndex 消息对应的已提交位置
                // 以及其消息 ID 封装成 ReadState 实例，添加到 raft.readStates 中保存。后续会有
                // 其他 goroutine 读取该数组，并对相应的 MsgReadIndex 消息进行响应
                r.readStates = append(r.readStates, ReadState{Index: rs.index,
                        RequestCtx: req.Entries[0].Data})
            } else {
                // 如果是其他 Follower 节点转发到 Leader 节点的 MsgReadIndex 消息，则 Leader 节点
                // 会向 Follower 节点返回相应的 MsgReadIndexResp 消息，并由 Follower 节点响应 Client
                r.send(pb.Message{To: req.From, Type: pb.MsgReadIndexResp,
                        Index: rs.index, Entries: req.Entries})
            }
        }
    }
}
```

上述处理过程中，首先会调用 readOnly.recvAck()方法统计指定的消息 ID 收到了多少 MsgHeartbeatResp 响应消息，从而确定在收到对应的 MsgReadIndex 消息的时候，当前 Leader



节点是否为集群的 Leader 节点，readOnly.recvAck()方法的具体实现片段如下：

```
func (ro *readOnly) recvAck(m pb.Message) int {
    // 获取消息 ID 对应的 readIndexStatus 实例，省略异常处理的代码
    rs, ok := ro.pendingReadIndex[string(m.Context)]
    rs.acks[m.From] = struct{}{} // 表示 MsgHeartbeatResp 消息的发送节点与当前节点连通
    return len(rs.acks) + 1 // 统计目前为止与当前节点连通的节点个数(最后+1 是当前节点自身)
}
```

当收到的响应消息超过半数以上时，会通过 readOnly.advance()方法清空指定消息 ID 及其之前的所有相关记录，advance()方法的具体实现如下：

```
func (ro *readOnly) advance(m pb.Message) []*readIndexStatus {
    var ( i int; found bool)
    ctx := string(m.Context) // MsgHeartbeat 消息对应的 MsgReadIndex 消息 ID
    rss := []*readIndexStatus{}
    for _, okctx := range ro.readIndexQueue { // 遍历 readOnly 中记录的消息 ID
        i++
        rs, ok := ro.pendingReadIndex[okctx]  // 查找消息 ID 对应的 readIndexStatus 实例，
                                              // 异常处理(略)
        rss = append(rss, rs) // 将 readIndexStatus 实例保存到 rss 数组中
        if okctx == ctx { // 查找到指定的 MsgReadIndex 消息 ID，则设置 found 变量并跳出循环
            found = true
            break
        }
    }

    if found { // 查找到指定的 MsgReadIndex 消息 ID，则清空 readOnly 中所有在它之前的消息 ID 及
               // 相关内容
        ro.readIndexQueue = ro.readIndexQueue[i:] // 清理 readOnly.readIndexQueue
        for _, rs := range rss { // 清理 readOnly.pendingReadIndex
            delete(ro.pendingReadIndex, string(rs.req.Entries[0].Data))
        }
        return rss
    }
    return nil
}
```



最后，我们再来简单了解一下 Follower 节点对 MsgReadIndex 消息和 MsgReadIndexResp 消息的处理。在集群中，Follower 节点不能直接响应客户端的只读请求，而是转发给 Leader 节点进行处理，等待 Leader 节点响应之后，Follower 节点才能响应 Client。raft.stepFollower()方法的相关代码片段如下：

```
func stepFollower(r *raft, m pb.Message) {
  switch m.Type {
  case pb.MsgReadIndex:
    if r.lead == None { // 检测当前集群中的 Leader 节点
      return
    }
    m.To = r.lead
    r.send(m) // 将 MsgReadIndex 消息转发给当前的 Leader 节点，raft.send()方法不再赘述
  case pb.MsgReadIndexResp:
    // 检测 MsgReadIndexResp 消息的合法性(略)
    // 将 MsgReadIndex 消息对应的消息 ID 以及已提交位置(raftLog.committed)封装成 ReadState
    // 实例，并添加到 raft.readStates 中，等待其他 goroutine 来处理
    r.readStates = append(r.readStates, ReadState{Index: m.Index,
      RequestCtx: m.Entries[0].Data})
  }
}
```

ReadOnlySafe 模式是 etcd 作者推荐的模式，在这种模式下的只读请求处理，不会受节点之间时钟差异和网络分区的影响，这也是本节到目前为止一直在介绍的模式。另一种模式是 ReadOnlyLeaseBased，虽然该模式并不推荐，但是其相关实现比较简单，读者也可以简单了解一下。stepLeader()方法中在 ReadOnlyLeaseBased 模式下处理 MsgReadIndex 消息的相关代码片段如下：

```
func stepLeader(r *raft, m pb.Message) {
  switch m.Type {
  case pb.MsgReadIndex: // 重点介绍 MsgReadIndex 消息相关代码片段，其他类型消息的代码片段省略
    if r.quorum() > 1 { // 集群场景
      // 相关检测(略)
      switch r.readOnly.option { // ReadOnlySafe 模式的相关代码片段(略)
      case ReadOnlyLeaseBased:
        var ri uint64
        if r.checkQuorum {
```



```
                // 检测是否开启了 CheckQuorum 模式，并据此更新返回给 Follower 的已提交位置
                ri = r.raftLog.committed
            }
            // ReadOnlyLeaseBased 模式下，Leader 节点直接并不会发送任何消息来确认自身身份
            if m.From == None || m.From == r.id {
                r.readStates = append(r.readStates,
                    ReadState{Index: r.raftLog.committed, RequestCtx: m.Entries[0].Data})
            } else {
                r.send(pb.Message{To: m.From,
                    Type: pb.MsgReadIndexResp, Index: ri, Entries: m.Entries})
            }
        }
    } else { // 单节点的情况下
        r.readStates = append(r.readStates,
                ReadState{Index: r.raftLog.committed, RequestCtx: m.Entries[0].Data})
    }
    return
  }
}
```

到此为止，MsgReadIndex 消息和 MsgReadIndexResp 消息的全部内容介绍完了。

➢ MsgSnap消息

通过前面介绍的 raft.sendAppend()方法可知，在 Leader 节点尝试向集群中的 Follower 节点发送 MsgApp 消息时，如果查找不到待发送的 Entry 记录（即该 Follow 节点对应的 Progress.Next 指定的 Entry 记录），则会尝试通过 MsgSnap 消息将快照数据发送到 Follower 节点，Follower 节点之后会通过快照数据恢复其自身状态，从而可以与 Leader 节点进行正常的 Entry 记录复制。例如，当 Follower 节点宕机时间比较长，就可能出现上述发送 MsgSnap 消息的场景。

首先看一下 Leader 节点发送 MsgSnap 消息的过程，相关的代码片段位于 raft.sendApp()方法中，具体实现如下：

```
func (r *raft) sendAppend(to uint64) {
  // 判断是否可以向指定的 Follower 节点发送消息(略)，创建 Message 实例(略)
  // 查找待发送的消息
  term, errt := r.raftLog.term(pr.Next - 1)
  ents, erre := r.raftLog.entries(pr.Next, r.maxMsgSize)
```



仅供非商业用途或交流学习使用


```
    if errt != nil || erre != nil { // 如果上述 raftLog 的查找出现异常，则会尝试发送 MsgSnap 消息
        if !pr.RecentActive { // 从当前的 Leader 节点的角度来看，目标 Follower 节点已经不再存活
            return
        }
        m.Type = pb.MsgSnap // 将消息类型设置成 MsgSnap，为后续发送快照数据做准备
        snapshot, err := r.raftLog.snapshot() // 获取快照数据
        // 异常检测，如果获取快照数据出现异常，则终止整个程序(略)
        m.Snapshot = snapshot // 设置 MsgSnap 消息的 Snapshot 字段
        sindex, sterm := snapshot.Metadata.Index, snapshot.Metadata.Term // 获取快照的
                                                                         // 相关信息

        // 将目标 Follower 节点对应的 Progress 切换成 ProgressStateSnapshot 状态，
        // 其中会用 Progress.PendingSnapshot 字段记录快照数据的信息
        pr.becomeSnapshot(sindex)
    } else {
        // 发送 MsgApp 消息的相关代码在前面已经介绍过了，这里不再赘述
    }
    r.send(m) // 发送 MsgSnap 消息，前面已经介绍 raft.senda()方法，这里不再赘述
}
```

下面转到 Follower 节点，介绍其对 MsgSnap 消息的处理，其相关的代码片段位于raft.stepFollower()方法中，具体实现如下：

```
func stepFollower(r *raft, m pb.Message) {
    switch m.Type { // 这里重点介绍 MsgSnap 消息的处理
    case pb.MsgSnap:
        r.electionElapsed = 0 // 重置 raft.electionElapsed，防止发生选举
        r.lead = m.From       // 设置 Leader 的 id
        r.handleSnapshot(m) // 通过 MsgSnap 消息中的快照数据，重建当前节点的 raftLog
    }
}
```

在 raft.handleSnapshot()方法中，会读取 MsgSnap 消息中的快照数据，并重建当前节点的raftLog，具体实现如下：

```
func (r *raft) handleSnapshot(m pb.Message) {
    // 获取快照数据的元数据
    sindex, sterm := m.Snapshot.Metadata.Index, m.Snapshot.Metadata.Term
```



仅供非商业用途或交流学习使用


```
    if r.restore(m.Snapshot) { // 返回值表示是否通过快照数据进行了重建
        // 向 Leader 节点返回 MsgAppResp 消息(Reject 始终为 false)。该 MsgAppResp 消息作为
        // MsgSnap 消息的响应，与前面介绍的 MsgApp 消息并无差别，Leader 节点对该 MsgAppResp 消息
        // 的处理也不再赘述
        r.send(pb.Message{To: m.From, Type: pb.MsgAppResp, Index: r.raftLog.lastIndex()})
    } else {
        r.send(pb.Message{To: m.From, Type: pb.MsgAppResp, Index: r.raftLog.committed})
    }
}
```

在 raft.restore()方法中，会调用 raftLog 中相应的方法进行检测和重建，其具体实现如下：

```
func (r *raft) restore(s pb.Snapshot) bool {
    // 根据快照数据的元数据查找匹配的 Entry 记录，如果存在，则表示当前节点已经拥有了该快照中的全部
    // 数据，所以无须进行后续重建操作
    if r.raftLog.matchTerm(s.Metadata.Index, s.Metadata.Term) {
        // 快照中全部的 Entry 记录都已经提交了，所以尝试修改当前节点的 raftLog.committed，
        // 通过前面的介绍我们知道，raftLog.committed 只增不减
        r.raftLog.commitTo(s.Metadata.Index)
        return false
    }

    // 通过 raftLog.unstable 记录该快照数据，同时重置相关字段
    r.raftLog.restore(s)
    // 清空 raft.prs，并根据快照的元数据进行重建
    r.prs = make(map[uint64]*Progress)
    for _, n := range s.Metadata.ConfState.Nodes {
        match, next := uint64(0), r.raftLog.lastIndex()+1
        if n == r.id { // 更新当前节点自身的 Match 值
            match = next - 1
        }
        r.setProgress(n, match, next) // 创建 Progress 实例，注意 Match 和 Next 的初始值
    }
    return true
}
```

这里还有两个本地消息与 MsgSnap 消息相关，它们分别是 MsgSnapStatus 消息和 MsgUnreachable 消息。如果 Leader 节点发送 MsgSnap 消息时出现异常，则会调用 Raft 接口的



仅供非商业用途或交流学习使用

ReportUnreachable()方法和 ReportSnapshot()方法发送 MsgUnreachable 消息和 MsgSnapStatus 消息。下面来看一下 raft.stepLeader()方法中关于 MsgUnreachable 消息和 MsgSnapStatus 消息的处理代码：

```
func stepLeader(r *raft, m pb.Message) {
    switch m.Type { // 这里重点介绍 MsgUnreachable 消息和 MsgSnapStatus 消息的相关代码
    case pb.MsgUnreachable:
        // 当 Follower 节点变得不可达，如果继续发送 MsgApp 消息，则会有大量消息丢失
        if pr.State == ProgressStateReplicate {
            pr.becomeProbe() // 将 Follower 节点对应的 Progress 实例切换成 ProgressStateProbe 状态
        }
    case pb.MsgSnapStatus:
        // 检测 Follower 节点对应的状态是否为 ProgressStateSnapshot(略)
        if !m.Reject { // 之前的发送 MsgSnap 消息时出现异常
            pr.becomeProbe()
        } else {
            // 发送 MsgSnap 消息失败，这里会清空对应 Progress.PendingSnapshot 字段
            pr.snapshotFailure()
            pr.becomeProbe()
        }
        // 无论 MsgSnap 消息发送是否失败，都会将对应 Progress 切换成 ProgressStateProbe 状态，
        // 之后单条发送消息进行试探

        // 暂停 Leader 节点向该 Follower 节点继续发送消息，如果发送 MsgSnap 消息成功了，则待 Leader 节点
        // 收到相应的响应消息(MsgAppResp 消息)，即可继续发送后续 MsgApp 消息，Leader 节点对 MsgAppResp
        // 消息的处理已经介绍过了，这里不再赘述；如果发送 MsgSnap 消息失败了，则 Leader 节点会等到
        // 收到 MsgHeartbeatResp 消息时，才会重新开始发送后续消息
        pr.pause()
    }
}
```

到此为止，etcd-raft 模块中关于快照复制的消息处理就介绍完了。

➢ MsgTransferLeader消息和MsgTimeoutNow消息

在前面的介绍中，我们都省略了对 Leader 节点转移的内容，本小节就重点分析 Leader 节点转移的过程，其中涉及 MsgTransferLeader 消息和 MsgTimeoutNow 消息的处理。Leader 节点转移一般是我们指定一个特定的节点作为下一个任期的 Leader 节点，当前 Leader 节点会选择一个



仅供非商业用途或交流学习使用

合适的节点（选择的条件后面详细介绍），然后发送 MsgTransferLeader 消息（本地消息）。

下面先来分析 Leader 节点对 MsgTransferLeader 消息的处理，stepLeader()方法中的代码片段如下：

```
func stepLeader(r *raft, m pb.Message) {
  switch m.Type { // 这里重点介绍 MsgTransferLeader 消息的处理，其他消息相关的代码片段暂时省略
  case pb.MsgTransferLeader:
    // 在 MsgTransferLeader 消息中，From 字段记录了此次 Leader 节点迁移操作的目标 Follower 节点 ID
    leadTransferee := m.From
    // 检测当前是否有一次未处理完的 Leader 节点转移操作
    lastLeadTransferee := r.leadTransferee
    if lastLeadTransferee != None {
      if lastLeadTransferee == leadTransferee {
        // 目标节点相同，则忽略此次 Leader 迁移操作
        return
      }
      r.abortLeaderTransfer() // 若目标节点不同,则清空上次记录的 ID(即 raft.leadTransferee)
    }
    if leadTransferee == r.id { // 目标节点已经是 Leader 节点，放弃此次迁移操作
      return
    }
    // Leader 迁移操作应该在 electionTimeout 时间内完成，这里会重置选举计时器
    r.electionElapsed = 0
    r.leadTransferee = leadTransferee // 记录此次 Leader 节点迁移的目标节点 ID
    // 检测目标 Follower 节点是否与当前 Leader 节点的 raftLog 是否完全一致
    if pr.Match == r.raftLog.lastIndex() {
      // 向目标 Follower 节点发送 MsgTimeoutNow 消息，这会导致 Follower 节点的选举计时器立即
      // 过期，并发起新一轮选举
      r.sendTimeoutNow(leadTransferee)
    } else {
      // 如果 raftLog 中的 Entry 记录没有完全匹配，则 Leader 节点通过发送 MsgApp 消息向目标节
      // 点进行复制
      r.sendAppend(leadTransferee)
    }
  }
}
```

在进行 Leader 节点转移时，Leader 节点会停止处理客户端发来的 MsgProp 消息，这是因为



仅供非商业用途或交流学习使用

如果客户端不断追加 Entry 记录，则可能使目标 Follower 节点与 Leader 节点的 raftLog 始终不匹配，从而导致 Leader 节点的迁移超时。这里回顾一下 Leader 节点处理 MsgProp 消息时与 Leader 节点迁移相关代码片段：

```
func stepLeader(r *raft, m pb.Message) {
    switch m.Type { // 这里只展示了 MsgProp 消息处理中 Leader 节点迁移的相关代码
    case pb.MsgProp:
        if r.leadTransferee != None { // 当前正在进行 Leader 节点转移操作，则不处理 MsgProp 消息
            return
        }
        // 向 raftLog 中追加 Entry 记录，以及发送 MsgApp 消息的代码片段(略)
        return
    }
}
```

下面我们转到目标 Follower 节点中，分析看看它是如何处理 MsgTimeoutNow 消息的。stepFollower()方法中相关的代码片段如下：

```
func stepFollower(r *raft, m pb.Message) {
    switch m.Type {
    case pb.MsgTransferLeader: // Follower 节点直接将 MsgTransferLeader 消息转给 Leader 节点
        m.To = r.lead
        r.send(m)
    case pb.MsgTimeoutNow:
        if r.promotable() { // 检测当前节点是否已被移出当前集群
            // 即使当前集群开启了 preVote 模式，目标 Follower 节点也会直接发起新一轮选举，
            // 后续选举的过程在前面已经介绍过了，这里不再赘述
            r.campaign(campaignTransfer)
        }
    }
}
```

这里需要读者注意的是，目标 Follower 节点在处理 MsgTimeoutNow 消息时调用的 raft.campaign()方法的参数是 campaignTransfer，该参数会被记录到 MsgVote 消息中并发送到集群中的其他节点，而其他节点在处理 MsgVote 消息时，会因为该标识立即参与此次选举，读者可以回顾前面对 MsgPreVote 消息和 MsgVote 消息处理过程的介绍。

另一个需要读者注意的地方是，前文提到当 Leader 节点与目标 Follower 节点的 raftLog 不



仅供非商业用途或交流学习使用

匹配时，会通过 MsgApp 消息进行 Entry 记录复制，直至两者完全匹配之后，才会向目标 Follower 节点发送 MsgTimeoutNow 消息，stepLeader()方法中相关的代码如下：

```
func stepLeader(r *raft, m pb.Message) {
  switch m.Type {
  case pb.MsgAppResp:
    pr.RecentActive = true // 回消息的节点活着

    if m.Reject { //
      // MsgApp 消息被拒绝的情况，这里不再赘述
    } else { // Leader 之前发送的 MsgApp 消息被 Follower 成功接收了
      // 更新对应的 Match 值(略)
      // 切换对应 Progress 的状态(略)
      // 检测是否有 Entry 记录已可以提交，并发送相应的消息(略)

      // 当收到目标 Follower 节点的 MsgAppResp 消息并且两者的 raftLog 完全匹配，
      // 则发送 MsgTimeoutNow 消息
      if m.From == r.leadTransferee && pr.Match == r.raftLog.lastIndex() {
        r.sendTimeoutNow(m.From)
      }
    }
  }
}
```

到这里，etcd-raft 模块中各种类型消息的含义及每种类型消息的处理流程就全部介绍完了，希望读者可以将本节内容与前面介绍的 Raft 协议结合在一起分析，这样可以更加清晰地了解 etcd-raft 中的实现。

3.2　Node接口

通过上一节对 raft 结构体及各种消息处理流程的分析可以看出，结构体 raft 实现了 Raft 协议中最核心的内容，它也是整个 etcd-raft 模块的核心，但是它并没有实现网络传输、持久化存储（注意与存储 Entry 记录的 raftLog 进行区分）等功能，也没有对外提供简单易用的 API。

在 etcd-raft 模块中，结构体 Node 表示集群中的一个节点，它是在结构体 raft 之上的一层封装，对外提供相对简单的 API 接口。下面看一下 Node 接口提供的方法：

```
type Node interface {
```



仅供非商业用途或交流学习使用


```
// 在前面介绍 Raft 算法及 raft 结构体时，提到了选举计时器和心跳计时器两种计时器，也提到过
// 这两个计时器的时间刻度是逻辑时间，并不是真实世界的时间刻度。Tick()方法就是用来推进逻辑时钟
// 的指针，从而推进上述的两个计时器
Tick()

// 当选举计时器超时之后，会调用 Campaign()方法会将当前节点切换成 Candidate 状态(或是 PerCandidate
// 状态)，底层就是通过发送 MsgHup 消息实现的，具体的选举(以及预选)流程参考上一小节的介绍
Campaign(ctx context.Context) error

// 接收到 Client 发来的写请求时，Node 实例会调用 Propose()方法进行处理，底层就是通过发送
// MsgProp 消息实现的，MsgProp 消息的处理流程参考上一小节的介绍
Propose(ctx context.Context, data []byte) error

// Client 除了会发送读写请求，还会发送修改集群配置的请求(例如，新增集群中的节点)，
// 这种请求 Node 实例会调用 ProposeConfChange()方法进行处理，底层就是通过发送
// MsgProp 消息实现的，只不过其中记录的 Entry 记录是 EntryConfChange 类型
ProposeConfChange(ctx context.Context, cc pb.ConfChange) error

// 当前节点收到其他节点的消息时，会通过 Step()方法将消息交给底层封装的 raft 实例进行处理
Step(ctx context.Context, msg pb.Message) error

// Ready()方法返回的是一个 Channel，通过该 Channel 返回的 Ready 实例中封装了底层 raft 实例的
// 相关状态数据，例如，需要发送到其他节点的消息、交给上层模块的 Entry 记录，等等。这是 etcd-raft
// 模块与上层模块交互的主要方式，Ready 的结构在后面会详细分析
Ready() <-chan Ready

// 当上层模块处理完从上述 Channel 中返回的 Ready 实例之后，需要调用 Advance()通知底层的
// etcd-raft 模块返回新的 Ready 实例
Advance()

// 在收到集群配置请求时，会通过调用 ApplyConfChange()方法进行处理
ApplyConfChange(cc pb.ConfChange) *pb.ConfState

// TransferLeadership()方法用于 Leader 节点的转移，具体实现后面详细介绍
TransferLeadership(ctx context.Context, lead, transferee uint64)

// ReadIndex()方法用于处理只读请求，具体实现后面详细介绍
ReadIndex(ctx context.Context, rctx []byte) error
```



仅供非商业用途或交流学习使用


```
    // Status()返回当前节点的状态运行状态，Status 结构体在后面详细介绍
    Status() Status

    // 通过 ReportUnreachable()方法通知底层的 raft 实例，当前节点无法与指定的节点进行通信
    ReportUnreachable(id uint64)

    // ReportSnapshot()方法用于通知底层的 raft 实例上次发送快照的结果
    ReportSnapshot(id uint64, status SnapshotStatus)

    Stop() // 关闭当前节点
}
```

在开始介绍 Node 接口中各方法的实现之前，有必要先介绍一下 Node 相关的几个结构体。首先来看一下 Ready 结构体，Ready 内嵌了 SoftState 和 HardState，在 SofteState 中封装了当前集群的 Leader 节点 ID（Lead 字段）及当前节点的角色（RaftState），在 HardState 中封装了当前节点的任期号（Term 字段）、当前节点在该任期投票结果（Vote 字段）及当前节点的 raftLog 的已提交位置。除此之外，Ready 实例中每个字段都封装一部分的数据，它们的含义如下所示。

- Entries（[]pb.Entry 类型）：该字段中的 Entry 记录是从 unstable 中读取出来的，上层模块会将其保存到 Storage 中。
- CommittedEntries（[]pb.Entry 类型）：已提交、待应用的 Entry 记录，这些 Entry 记录之前已经保存到了 Storage 中。
- Snapshot（pb.Snapshot 类型）：待持久化的快照数据，raftpb.Snapshot 中封装了快照数据及相关元数据，在前面已经详细介绍过了，这里不再重复介绍。
- Messages（[]pb.Message 类型）：该字段中保存了当前节点中等待发送到集群其他节点的 Message 消息。
- ReadStates（[]ReadState 类型）：该字段记录了当前节点中等待处理的只读请求。

最后需要注意的是，Ready 实例只是用来传递数据的，其全部字段都是只读的。

Status 是另一个需要介绍的结构体，它也内嵌了 HardState 和 SoftState，除此之外，它还记录了当前节点的 ID（ID 字段）、已应用的 Entry 记录的最大索引值（Applied 字段）。

如果当前节点是 Leader 节点在其 Progress 字段（map[uint64]Progress 类型）中还记录了集群中每个节点对应的 Progress 实例，raft.Progress 结构体在前面已经介绍过了，这里不再重复介绍。



仅供非商业用途或交流学习使用

3.2.1 node结构体

node 结构体是 Node 接口的实现之一，本小节重点介绍 node 的具体实现。这里首先来分析 node 结构体中各个字段的含义，它提供的方法在后面的分析过程中详细介绍。

- propc（chan pb.Message 类型）：该通道用于接收 MsgProp 类型的消息。
- recvc（chan pb.Message 类型）：除 MsgProp 外的其他类型的消息都是由该通道接收的。
- confc（chan pb.ConfChange 类型）：当节点收到 EntryConfChange 类型的 Entry 记录时，会转换成 ConfChange，并写入该通道中等待处理。在 ConfChange 中封装了其唯一 ID、待处理的节点 ID（NodeID 字段）及处理类型（Type 字段，例如，ConfChangeAddNode 类型表示添加节点）等信息。
- confstatec（chan pb.ConfState 类型）：在 ConfState 中封装了当前集群中所有节点的 ID，该通道用于向上层模块返回 ConfState 实例。
- readyc（chan Ready 类型）：Ready 结构体的功能在上一小节已经介绍过了，该通道用于向上层模块返回 Ready 实例，即 node.Ready()方法的返回值。
- advanced（chan struct{}类型）：当上层模块处理完通过上述 readyc 通道获取到的 Ready 实例之后，会通过 node.Advance()方法向该通道写入信号，从而通知底层 raft 实例。
- status（chan chan Status 类型）：Status 结构体在上一小节已经介绍过了，注意该通道的类型，其中传递的元素也是 Channel 类型，即 node.Status()方法的返回值。
- tickc（chan struct{}类型）：用来接收逻辑时钟发出的信号，之后会根据当前节点的角色推进选举计时器和心跳计时器。
- stop（chan struct{}类型）：当 node.Stop()方法被调用时，会向该通道发送信号，在后续介绍中会提到，有另一个 goroutine 会尝试读取该通道中的内容，当读取到信息之后，会关闭 done 通道。
- done（chan struct{}类型）：当检测到 done 通道关闭后，在其上阻塞的 goroutine 会继续执行，并进行相应的关闭操作。

3.2.2 初始化

当集群中的节点初次启动时会通过 StartNode()函数启动创建对应的 node 实例和底层的 raft 实例。在 StartNode()方法中，主要是根据传入的 Config 配置创建 raft 实例并初始化 raft 使用的相关组件，其具体实现如下：



```
// 首先介绍一下第二个参数，Peer 封装了节点的 ID，peers 记录当前集群中全部节点的 ID
func StartNode(c *Config, peers []Peer) Node {
    r := newRaft(c) // 根据 Config 配置信息创建 raft 实例
    r.becomeFollower(1, None) // 切换成 Follower 状态，因为节点初次启动，所以任期号为 1
    for _, peer := range peers {
        // 根据传递的节点列表，创建对应的 ConfChange 实例，其 Type 是 ConfChangeAddNode，
        // 表示添加指定节点
        cc := pb.ConfChange{Type: pb.ConfChangeAddNode, NodeID: peer.ID,
                Context: peer.Context}
        d, err := cc.Marshal() // 序列化 ConfChange 记录
        // 将 ConfChange 记录序列化后的数据封装成 EntryConfChange 类型的 Entry 记录
        e := pb.Entry{Type: pb.EntryConfChange, Term: 1,
                Index: r.raftLog.lastIndex() + 1, Data: d}
        r.raftLog.append(e) // 将 EntryConfChange 记录追加到 raftLog 中
    }
    // 直接修改 raftLog.committed 值，提交上述 EntryConfChange 记录
    r.raftLog.committed = r.raftLog.lastIndex()

    // 为节点列表中的每个节点都创建对应的 Progress 实例
    for _, peer := range peers {
        r.addNode(peer.ID) // 添加节点
    }
    n := newNode() // 初始化 node 实例
    n.logger = c.Logger
    // 启动一个 goroutine，其中会根据底层 raft 的状态及上层模块传递的数据，
    // 协调处理 node 中各种通道的数据
    go n.run(r)
    return &n
}
```

newRaft()函数会完成对底层 raft 实例的初始化，在前面已经详细介绍过了，这里不再重复描述。在上面的 newNode()函数中，会完成 node 实例的初始化，其中会初始化上面介绍的全部通道，具体实现如下：

```
func newNode() node {
    return node{
        propc:      make(chan pb.Message),
        recvc:      make(chan pb.Message),
```



```
        confc:      make(chan pb.ConfChange),
        confstatec: make(chan pb.ConfState),
        readyc:     make(chan Ready),
        advancec:   make(chan struct{}),
        tickc:  make(chan struct{}, 128), // 带缓冲的 Channel
        done:   make(chan struct{}),
        stop:   make(chan struct{}),
        status: make(chan chan Status),
    }
}
```

当集群中的节点重新启动时，就不是通过 StartNode()函数创建 node 实例，而是调用 RestartNode()函数，RestartNode()函数与 StartNode()函数的实现非常类似，其中最主要的区别就是不会根据 Config 中的配置信息调用 raft.addNode()方法添加节点，因为这些信息会从 Storage 中恢复。RestartNode()函数的具体实现如下：

```
func RestartNode(c *Config) Node {
    r := newRaft(c) // 创建底层封装的 raft 实例，该过程与 StartNode()函数相同
    n := newNode()  // 创建 node 实例
    // 启动一个 goroutine，其中会根据底层 raft 的状态及上层模块传递的数据，
    // 协调处理 node 中各种通道的数据
    go n.run(r)
    return &n
}
```

3.2.3 run()方法

node.run()方法会处理 node 结构体中封装的全部通道，它也是 node 的核心。正是因为在该方法中会对多个通道进行处理，所以本小节会将其中几个比较重要的通道处理过程单独拿出来介绍。node.run()方法的大致实现如下：

```
func (n *node) run(r *raft) {
    var propc chan pb.Message // 指向 node.propc 通道
    var readyc chan Ready // 指向 node.readyc 通道
    var advancec chan struct{} // 指向 node.advancec 通道
    // 用于记录 unstable 中最后一条 Entry 记录的索引值和 Term 值
    var prevLastUnstablei, prevLastUnstablet uint64
```



```
var havePrevLastUnstablei bool
var prevSnapi uint64 // 用于记录快照中最后一条记录的索引值
var rd Ready

lead := None   // 用于记录当前 Leader 节点
prevSoftSt := r.softState() // 用于记录 SoftState(当前 Leader 节点 ID 和当前节点的角色)
prevHardSt := emptyState // 用于记录 HardState

for { // 在该循环中完成了对所有相关 Channel 的处理
  if advancec != nil {
    // 上层模块还在处理上次从 readyc 通道返回的 Ready 实例，所以不能继续向 readyc 中写入数据
    readyc = nil
  } else {
    rd = newReady(r, prevSoftSt, prevHardSt) // 创建 Ready 实例
    // 根据 Ready 决定，是否存在需要交给上层模块处理的数据
    if rd.containsUpdates() {
      readyc = n.readyc
    } else {
      readyc = nil // 此次创建的 Ready 实例中各个字段都为空，无须交给上层模块处理
    }
  }

  if lead != r.lead { // 检测当前的 Leader 节点是否发生变化
    // 如果当前节点无法确定集群中的 Leader 节点，则清空 propc，此次循环不再处理 MsgPropc 消息
    if r.hasLeader() {
      propc = n.propc
    } else {
      propc = nil
    }
    lead = r.lead // 更新 leader
  }

  select {
  case m := <-propc: // 读取 propc 通道，获取 MsgPropc 消息，并交给 raft.Step()方法处理
    m.From = r.id
    r.Step(m)  // raft.Step()方法在上一小节已经详细分析过了，这里不再赘述
  case m := <-n.recvc: // 读取 node.recvc 通道，获取消息(非 MsgPropc 类型)，
                       // 并交给 raft.Step()方法进行处理
```



```
            // 如果是来自未知节点的响应消息(例如，MsgHeartbeatResp 类型消息)则会被过滤
            if _, ok := r.prs[m.From]; ok || !IsResponseMsg(m.Type) {
                r.Step(m)
            }
        case <-n.tickc: // 逻辑时钟每推进一次，就会向 tickc 通道写入一个信号
            // raft.tick()在前面已经详细介绍过，Leader 节点推进选举计时器，Follower 节点推进心跳
            // 计时器，这里不再赘述
            r.tick()
        case readyc <- rd: // 将前面创建的 Ready 实例写入 node.readyc 通道中，等待上层模块读取
        ... ...
        case <-advancec: // 上层模块处理完 Ready 实例之后，会向 advance 通道写入信号
        ... ...
        case cc := <-n.confc: // 读取 node.confc 通道，获取 ConfChange 实例，下面根据 ConfChange
                              // 中的类型进行分类处理
        ... ...
        case c := <-n.status:
        ... ...
        case <-n.stop:
            // 当从 stop 通道中读取到信号时，会将 done 通道关闭，阻塞在其上的 goroutine 可以继续执行
            close(n.done)
            return
        }
    }
}
```

首先看一下 newReady()函数，它会读取底层 raft 实例中的各项数据及相关状态，并最终封装成 Ready 实例，该 Ready 实例最终会返回给上层模块，具体实现如下：

```
// 解析一些该函数中几个参数的含义：prevSoftSt、prevHardSt 是上次创建 Ready 实例时记录的 raft 实
// 例的相关状态
func newReady(r *raft, prevSoftSt *SoftState, prevHardSt pb.HardState) Ready {
    rd := Ready{
        // 获取 raftLog 中 unstable 部分存储的 Entry 记录，这些 Entry 记录会交给上层模块进行持久化
        Entries:          r.raftLog.unstableEntries(),
        // 获取已提交但未应用的 Entry 记录，即 raftLog 中 applied~committed 之间的所有记录
        CommittedEntries: r.raftLog.nextEnts(),
        // 获取待发送的消息，在上一节介绍 raft 实例处理消息的相关代码可以看到，最终所有待发送的消息
        // 都会记录到 raft 实例的 msgs 字段中
```



```
        Messages:         r.msgs,
    }
    // 检测两次创建 Ready 实例之间，raft 实例状态是否发生变化，如果无变化，则将 Ready 实例相关字
    // 段设置为 nil，表示无须上层模块处理，SoftState 和 HardState 的比较方法比较简单，这里不再赘述
    if softSt := r.softState(); !softSt.equal(prevSoftSt) {
        rd.SoftState = softSt
    }
    if hardSt := r.hardState(); !isHardStateEqual(hardSt, prevHardSt) {
        rd.HardState = hardSt
    }

    // 检测 unstable 中是否记录了新的快照数据，如果有，则将其封装到 Ready 实例中，交给上层模块进行处理
    if r.raftLog.unstable.snapshot != nil {
        rd.Snapshot = *r.raftLog.unstable.snapshot
    }
    // 在前面介绍只读请求的处理时，raft 最终会将能响应的请求信息封装成 ReadState 并记录到
readStates 中，
    // 这里会检测 raft.readStates 字段，并将其封装到 Ready 实例返回给上层模块
    if len(r.readStates) != 0 {
        rd.ReadStates = r.readStates
    }
    return rd
}
```

通过 newReady()方法得到 Ready 实例之后，会通过 containsUpdates()方法检测 Ready 实例中各个字段是否为空，从而决定此次创建的 Ready 实例是否有必要交给上层模块处理进行处理。containsUpdates()方法的具体实现如下：

```
func (rd Ready) containsUpdates() bool {
    return rd.SoftState != nil || !IsEmptyHardState(rd.HardState)// 检测 raft 状态是否
                                                                 // 发生变化
        ||!IsEmptySnap(rd.Snapshot)  // 检测是否有新的快照数据
        || len(rd.Entries) > 0       // 检测是否有待持久化的 Entry 记录
        || len(rd.CommittedEntries) > 0  // 检测是否有待应用的 Entry 记录
        || len(rd.Messages) > 0      // 检测是否有待发送的消息
        || len(rd.ReadStates) != 0   // 检测是否有处理的只读请求
}
```



通过上述 containsUpdates()方法确定需要将 Ready 实例返回给上层模块之后，会将相应的 Ready 实例写入 readyc 通道中，该过程的相关片段如下：

```
func (n *node) run(r *raft) {
  // 相关变量的含义及初始化(略)
  for {
    // 创建 Ready 实例并进行检测(略)，检测当前集群的 Leader 节点并设置 propc(略)
    select {
    case readyc <- rd: // 将前面创建的 Ready 实例写入 node.readyc 通道中，等待上层模块读取
      if rd.SoftState != nil {
        prevSoftSt = rd.SoftState  // prevSoftSt 记录此次返回 Ready 实例的 SoftState 状态
      }
      if len(rd.Entries) > 0 {
        // 记录此次返回的 Ready 实例是否包含待持久化的 Entry，并记录最后一条记录的索引值和 Term 值
        prevLastUnstablei = rd.Entries[len(rd.Entries)-1].Index
        prevLastUnstablet = rd.Entries[len(rd.Entries)-1].Term
        havePrevLastUnstablei = true
      }
      if !IsEmptyHardState(rd.HardState) {
        prevHardSt = rd.HardState // prevHardSt 记录此次返回的 Ready 实例的 HardState 状态
      }
      if !IsEmptySnap(rd.Snapshot) { // prevSnapi 记录了此次返回的 Ready 实例中快照的元数据
        prevSnapi = rd.Snapshot.Metadata.Index
      }
      r.msgs = nil // 清空 raft.msgs 和 raft.readStates
      r.readStates = nil
      // 将 advancec 指向 node.advancec 通道，这样在下次 for 循环时，就无法继续向上层模块返回
      // Ready 实例了(因为 readyc 会被设置为 nil，无法向 readyc 通道中写入 Ready 实例)
      advancec = n.advancec
    }
  }
}
```

通过上面的分析可知，在向 readyc 通道写入 Ready 实例之后，会使用 pre*变量记录 Ready 实例中的相关状态信息，在下次创建 Ready 实例时，会根据这些 pre*变量判断当前底层的 raft 实例与上次相比是否已经发生了变化，并由此决定是否设置 Ready 实例的相应字段。

当上层模块通过 readyc 通道读取 Ready 实例之后，会将其中封装数据进行一系列处理，例



如，上层模块会应用 Ready.CommittedEntries 中的 Entry 记录、持久化 Ready.Entries 中的记录和快照数据等。当上层模块处理完此次返回的 Ready 实例之后，会通过 node.Advance()方法向 node.advancec 通道写入信号，通知 etcd-raft 模块其此次返回的 Ready 实例已经被处理完成，具体实现如下：

```
func (n *node) Advance() {
  select {
  case n.advancec <- struct{}{}: // 向 advancec 通道写入信号
  case <-n.done: // 当前 node 已关闭
  }
}
```

下面回到 node.run()方法中介绍 advancec 通道的相关处理，其中会根据 pre*记录的状态更新 raftLog 的相关字段，具体实现如下：

```
func (n *node) run(r *raft) {
  // 相关变量的含义及初始化(略)
  for {
    // 创建 Ready 实例并进行检测(略)，检测当前集群的 Leader 节点并设置 propc(略)
    select {
    case <-advancec: // 当读取到 advancec 通道中的信号时，则表示上层模块已经处理完 Ready 实例
      if prevHardSt.Commit != 0 { // 更新 raftLog.applied 字段(即已应用的记录位置)
        r.raftLog.appliedTo(prevHardSt.Commit)
      }
      if havePrevLastUnstablei {
        // Ready.Entries 中的记录已经被持久化了，这里会将 unstable 中的对应记录清理掉，
        // 并更新其 offset，也可能缩小 unstable 底层保存 Entry 记录的数组
        r.raftLog.stableTo(prevLastUnstablei, prevLastUnstablet)
        havePrevLastUnstablei = false
      }
      // Ready 中封装的快照数据已经被持久化，这里会清空 unstable 中记录的快照数据
      r.raftLog.stableSnapTo(prevSnapi)
      advancec = nil // 清空 advancec，下次 for 循环处理的过程中，就能向 readyc 通道写入 Ready 了
    }
  }
}
```

当上层模块在处理待应用的 Entry 记录（即 Ready.CommittedEntries 中保存的记录）时，会



对 Entry 记录进行分类处理，上层模块会将 EntryNormal 类型的记录（对应一个普通的数据写入操作）应用到自身状态机中，而对于 EntryConfChange 类型的记录（对应一个集群配置更改操作，例如，添加或删除一个节点），上层模块会将其封装成 ConfChange 实例并写入 node.confc 通道中，等待底层的 raft 实例进行处理。

上层模块处理 Entry 记录的过程在后面详细分析，这里我们重点分析 node.run()方法中对 node.confc 通道中 ConfChange 实例的处理，相关代码片段如下：

```
func (n *node) run(r *raft) {
  for { // 在该循环中完成了对所有相关 Channel 的处理
    select {
    case cc := <-n.confc: // 读取 node.confc 通道，获取 ConfChange 实例，下面根据 ConfChange
                          // 中的类型进行分类处理
      if cc.NodeID == None { // ConfChange 中未指定待处理的节点 ID
        r.resetPendingConf() // 重置 pendingConf 字段为 false，表示不存在待处理的节点变更请求
        select {
        // 将当前集群中的节点信息封装成 ConfState 实例并写入 confstatec 通道中
        case n.confstatec <- pb.ConfState{Nodes: r.nodes()}:
        case <-n.done:
        }
        break
      }
      switch cc.Type {
      case pb.ConfChangeAddNode:    // 添加节点，raft.addNode()方法的实现在后面详细分析
        r.addNode(cc.NodeID)
      case pb.ConfChangeRemoveNode: // 删除节点，raft.removeNode()方法的实现在后面详细分析
        if cc.NodeID == r.id {
          propc = nil // 在删除当前节点之前，首先会清空 propc 变量，保证后续不再处理 MsgPropc 消息
        }
        r.removeNode(cc.NodeID)
      case pb.ConfChangeUpdateNode:  // 更新节点
        r.resetPendingConf()  // 重置 pendingConf 字段为 false
      default:
        panic("unexpected conf type") // 未知的 ConfChange 类型
      }
      select {
      // 将当前集群中节点的信息封装成 ConfState 实例并写入 confstatec 通道中，上层模块会读取
      // 该通道，并从中获取当前集群的节点信息
```



```
            case n.confstatec <- pb.ConfState{Nodes: r.nodes()}:
            case <-n.done:
            }
        }
    }
}
```

在前面介绍 raft 结构体时并没有详细分析其如何完成节点增删操作，下面分析 rafte.addNode()方法，具体实现如下：

```
func (r *raft) addNode(id uint64) {
    r.pendingConf = false // 重置 pendingConf 字段
    if _, ok := r.prs[id]; ok { // 如果重复添加同一节点则忽略
        return
    }
    r.setProgress(id, 0, r.raftLog.lastIndex()+1) // 为新增节点创建对应的 Progress 实例
    // 认为新增节点与当前节点连通，否则在进行 CheckQuorum 检测时可能会使得当前节点放弃其 Leader 角色
    r.prs[id].RecentActive = true
}
```

再来看一下 raft.removeNode()方法是如何实现节点删除的，如下：

```
func (r *raft) removeNode(id uint64) {
    r.delProgress(id) // 从 prs 字段中删除对应的 Progress 实例
    r.pendingConf = false // 重置 pendingConf 字段
    if len(r.prs) == 0 { // 让集群中全部节点都被清空时直接返回
        return
    }
    // 当集群减小时，可能某些之前未提交的消息，现在可以提交了(之前已复制的节点个数未到半数，但是，
    // 因为集群的总节点数减少了，所以可能会到达半数以上)
    if r.maybeCommit() {
        r.bcastAppend() // 当有记录提交时，向集群其他节点发送消息，修改已提交位置
    }
    // 如果在 Leader 节点转移的过程中将目标节点删除，则放弃此次 Leader 节点转移
    if r.state == StateLeader && r.leadTransferee == id {
        r.abortLeaderTransfer()
    }
}
```



最后，我们来看一下 run()方法中对于 n.status 通道的处理，这里要注意一下，node.status 通道中传递的是 chan Status 类型的通道，相关代码片段如下：

```
func (n *node) run(r *raft) {
  // 相关变量的含义及初始化(略)
  for {
    // 创建 Ready 实例并进行检测(略)，检测当前集群的 Leader 节点并设置 propc(略)
    select {
    case c := <-n.status:
      c <- getStatus(r) // 创建 Status 实例
  }
}
```

node 对外提供了 Status()方法，该方法用于读取当前节点的状态，其实现如下：

```
func (n *node) Status() Status {
  c := make(chan Status) // 创建一个用于传递 Status 实例的通道
  select {
  case n.status <- c: // 将该通道写入到 node.status 通道中
    return <-c        // 阻塞等待 run()方法写入 Status 实例
  // 省略对 done 通道的处理
  }
}
```

下面回到 node.run()方法中，分析一下其中调用的 getStatue()函数是如何将当前节点的状态封装成 Status 实例的，具体实现如下：

```
func getStatus(r *raft) Status {
  s := Status{ID: r.id} // 创建 Status 实例，其 ID 字段为当前节点的 id
  s.HardState = r.hardState()     // 从底层的 raft 实例中获取 HardState 实例
  s.SoftState = *r.softState()    // 从底层的 raft 实例中获取 SoftState 实例
  s.Applied = r.raftLog.applied   // 获取底层的 raftLog 中记录已应用的位置

  // 如果当前节点是 Leader 状态，则将集群中每个节点对应的 Progress 实例封装到 Status 实例中
  if s.RaftState == StateLeader {
    s.Progress = make(map[uint64]Progress)
    for id, p := range r.prs {
      s.Progress[id] = *p
```



```
        }
    }
    return s
}
```

node.run()方法中对核心通道的处理过程就介绍到这里。

3.2.4　Node接口实现

介绍完 node 中最核心的 run()方法之后，我们来看一下 node 结构体实现的 Node 接口方法。node 实现的多个 Node 接口方法中，例如，Propose()方法、Campaign()方法、ReadIndex()方法和 ProposeConfChange()方法，都是通过调用 node.step()方法实现的，只是传入底层 raft 实例的消息类型有所不同，从而实现不同功能，上述四个方法对应传入底层的消息类型分别是 MsgProp、MsgHup、MsgReadIndex 和 MsgProp（其中只有一条 EntryConfChange 类型的 Entry 记录）。这四个方法的实现比较简单，代码就不贴出来了，感兴趣的读者可以参考源码进行学习。

node.Step()方法也是通过调用 node.step()方法实现的，该方法主要处理从其他节点收到的网络消息，而不是本地消息。下面具体分析：

```
func (n *node) step(ctx context.Context, m pb.Message) error {
    // 根据传入的消息类型，选择合适的通道
    ch := n.recvc
    if m.Type == pb.MsgProp {
        ch = n.propc
    }
    select {
    case ch <- m: // 将消息写入 recvc 或 propc 通道中
        return nil
    // 异常情况(略)
}
```

读者可以回顾上一小节对 node.run()方法的介绍，当其收到 recvc 通道或是 propc 通道写入的消息之后，会调用 raft.Step()方法将消息传递给底层的 raft 实例进行处理，相关的实现在前面已经介绍过了，这里不再赘述。

在前面介绍 MsgSnap 消息时提到过，当 Leader 节点发送 MsgSnap 消息时，如果出现异常，则会通过 Raft 接口的 ReportUnreachable()方法和 ReportSnapshot()方法发送 MsgUnreachable 消息和 MsgSnapStatus 消息。在 Raft 接口的实现中，上述两个方法是通过调用 node 实例的同名方法



实现的。在 node.ReportUnreachable()和 ReportSnapshot()方法中，分别会创建 MsgUnreachable 和 MsgSnapStatus 消息，然后直接写入 node.recvc 通道中等待处理，具体实现如下：

```
func (n *node) ReportSnapshot(id uint64, status SnapshotStatus) {
    rej := status == SnapshotFailure  // 用于设置 MsgSnapStatus 消息的 Reject
    select {
    case n.recvc <- pb.Message{Type: pb.MsgSnapStatus, From: id, Reject: rej}:
    case <-n.done:
    }
}
// ReportUnreachable()方法的实现类似，代码就不再贴出来了
```

除此之外，node.TransferLeadership()方法的实现也是通过创建相应类型的消息（MsgTransferLeader 类型消息）并写入 node.recvc 通道实现的，其代码比较简单，不再粘贴出来，感兴趣的读者可以参考源码进行学习。

到此为止，etcd-raft 模块的核心实现就全部介绍完了，在下一节，我们将通过一个示例，分析 etcd-raft 模块是如何使用的，也为后面介绍分析 etcd 中的其他模块打下基础。

3.3 raftexample示例分析

在 etcd 的源码中，自带了一个 raftexample 示例，在为数不多的资料中，该示例算是学习使用 etcd-raft 模块的最佳实践之一了。在本节中，我们就通过该示例深入分析如何使用 etcd-raft 模块，也可以让读者了解 etcd-raft 模块是如何与其他模块进行协作的。

在开始介绍 raftexample 示例之前，我们先通过图 3-12 了解一下 raftexample 示例的总体架构。

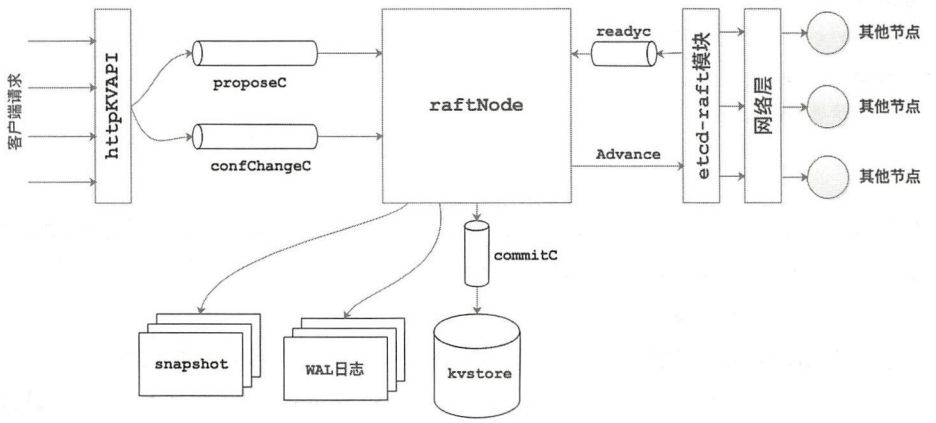


图 3-12



raftexample 示例的代码在 etcd 源码的/contrib/raftexample 目录中，其中比较核心的组件如下。

- raftNode：rafeNode 是该示例的核心组件（它定义在 raftexample/raft.go 文件中），raftNode 是上层模块和底层 etcd-raft 组件之间衔接的桥梁。raftNode 是对上一小节介绍的 etcd-raft 模块的一层封装，对上层模块提供了更加简洁、更方便使用的调用方式，可以让示例中其他部分无须过多关注 etcd-raft 模块的实现细节，降低了系统耦合程度。除此之外，raftNode 还封装了很多其他的功能，例如，前面提到的 WAL 日志管理、快照管理及网络层相关的功能。在该示例中，raftNode 相对于 etcd-raft 模块来说，扮演了上层模块的角色。
- kvstore：用于存储键值对信息，kvstore 扮演了持久化存储的角色。
- httpKVAPI：该示例向外提供的是 HTTP 接口，用户可以通过调用 HTTP 接口来模拟客户端的行为。在后面会有专门的一章详细介绍 etcd 提供的客户端。

3.3.1　raftNode

正如上一小节对 raftNode 定位的介绍，它底层除了封装了 etcd-raft 模块，还封装了很多其他相关组件，在 raftNode 中主要实现的功能如下：

- 将客户端发来的请求传递给底层 etcd-raft 组件中进行处理。
- 从 node.readyc 通道中读取 Ready 实例，并处理其中封装的数据，具体的处理过程在后面详细介绍。
- 管理 WAL 日志文件。
- 管理快照数据。
- 管理逻辑时钟。
- 将 etcd-raft 模块返回的待发送消息通过网络组件发送到指定的节点。

下面详细介绍 raftNode 结构体中核心字段的含义。

- proposeC (<–chan string 类型)：在 raftexample 示例中，HTTP PUT 请求表示添加键值对数据，当收到 HTTP PUT 请求时，httpKVAPI 会将请求中的键值信息通过 proposeC 通道传递给 raftNode 实例进行处理。
- confChangeC(<–chan raftpb.ConfChange 类型): 在 raftexample 示例中，HTTP POST 请求表示集群节点修改的请求，当收到 POST 请求时，httpKVAPI 会通过 confChangeC 通道将修改的节点 ID 传递给 raftNode 实例进行处理。
- commitC (chan<– *string 类型)：在创建 rafNode 实例之后（raftNode 实例的创建过程是在 newRaftNode()函数中完成的，其具体实现在后面详细介绍）会返回 commitC、



errorC、snapshotterReady 三个通道。raftNode 会将 etcd-raft 模块返回的待应用 Entry 记录（封装在 Ready 实例中）写入 commitC 通道，另一方面，kvstore 会从 commitC 通道中读取这些待应用的 Entry 记录并保存其中的键值对信息。

- errorC (chan<- error 类型)：当 etcd-raft 模块关闭或是出现异常的时候，会通过 errorC 通道将该信息通知上层模块。
- snapshotterReady (chan *snap.Snapshotter 类型)：主要用于初始化的过程中监听 snapshotter 实例是否创建完成，snapshotter 负责管理 etcd-raft 模块产生的快照数据，后面会详细介绍。
- id (int 类型)：记录当前节点的 ID。
- peers ([]string 类型)：当前集群中所有节点的地址，当前节点会通过该字段中保存的地址向集群中其他节点发送消息。
- join (bool 类型)：当前节点是否为后续加入到一个集群的节点。
- waldir (string 类型)：存放 WAL 日志文件的目录。
- wal (*wal.WAL 类型)：负责 WAL 日志的管理。当节点收到一条 Entry 记录时，首先会将其保存到 raftLog.unstable 中，之后会将其封装到 Ready 实例中并交给上层模块发送给集群中的其他节点，并完成持久化。在 raftexample 示例中，Entry 记录的持久化是将其写入 raftLog.storage 中。在持久化之前，Entry 记录还会被写入 WAL 日志文件中，这样就可以保证这些 Entry 记录不会丢失。WAL 日志文件是顺序写入的，所以其写入性能不会影响节点的整体性能。
- snapdir (string 类型)：存放快照文件的目录。
- snapshotter (*snap.Snapshotter 类型)：负责管理快照数据，etcd-raft 模块并没有完成快照数据的管理，而是将其独立成一个单独的模块，在后面有单独的章节来介绍相关模块。
- snapshotterReady (chan *snap.Snapshotter 类型)：该通道用于通知上层模块 snapshotter 实例是否已经创建完成。
- snapCount (uint64 类型)：两次生成快照之间间隔的 Entry 记录数，即当前节点每处理一定数量的 Entry 记录，就要触发一次快照数据的创建。每次生成快照时，即可释放掉一定量的 WAL 日志及 raftLog 中保存的 Entry 记录，从而避免大量 Entry 记录带来的内存压力及大量的 WAL 日志文件带来的磁盘压力；另外，定期创建快照也能减少了节点重启时回放的 WAL 日志数量，加速了启动时间。
- getSnapshot (func() ([]byte, error))：用于获取快照数据的函数，在 raftexample 示例中，该函数会调用 kvstore.getSnapshot()方法获取 kvstore.kvStore 字段的数据，后面还



会提到该函数的功能。

- lastIndex（uint64 类型）：当回放 WAL 日志结束之后，会使用使用该字段记录最后一条 Entry 记录的索引值。
- confState（raftpb.ConfState 类型）：用于记录当前的集群状态，该状态就是从前面介绍的 node.confstatec 通道中获取的。
- snapshotIndex（uint64 类型）：保存当前快照的相关元数据，即快照所包含的最后一条 Entry 记录的索引值。
- appliedIndex（uint64 类型）：保存上层模块已应用的位置，即已应用的最后一条 Entry 记录的索引值。
- node（raft.Node 类型）：即前面介绍的 etcd-raft 模块中的 node 实例，它实现了 Node 接口，并将 etcd-raft 模块的 API 接口暴露给了上层模块。
- raftStorage（*raft.MemoryStorage 类型）：前面已经介绍过 Storage 接口及其具体实现 MemoryStorage，这里不再重复介绍。在 raftexample 示例中，该 MemoryStorage 实例与底层 raftLog.storage 字段指向了同一个实例，在后面介绍 raftNode 初始化时还会提到该实例的初始化过程。
- transport（*rafthttp.Transport 类型）：通过前面介绍可知，节点待发送的消息只是记录到了 raft.msgs 中，etcd-raft 模块并没有提供网络层的实现，而由上层模块决定两个节点之间如何通信。这样就为网络层的实现提供了更大的灵活性，例如，如果两个节点在同一台服务器中，我们完全可以使用共享内存方式实现两个节点的通信，并不一定非要通过网络设备完成。在后面会有单独的章节介绍 etcd-rafthttp 模块是如何实现集群内消息的发送。
- httpstopc 和 httpdonec（chan struct{}类型）：这两个通道相互协作，完成当前节点的关闭工作，两者的工作方式与前面介绍的 node.done 和 node.stop 的工作方式类似，在后面介绍 raftexample 示例中的 stoppableListener 时再详细介绍。

了解了 raftNode 的主要功能及其各个字段的含义之后，我们开始分析 newRaftNode()函数，在该函数中主要完成了 raftNode 的初始化。在该方法中会使用上层模块传入的配置信息（其中包括 proposeC 通道和 confChangeC 通道）来创建 raftNode 实例，同时会创建 commitC 通道和 errorC 通道返回给上层模块使用。这样，上层模块就可以通过这几个通道与 rafeNode 实例进行交互了。另外，newRaftNode()函数中还会启动一个独立的后台 goroutine 来完成回放 WAL 日志、启动网络组件等初始化操作。newRaftNode()函数的具体实现如下：

```
func newRaftNode(id int, peers []string, join bool, getSnapshot func() ([]byte, error),
  proposeC <-chan string, confChangeC <-chan raftpb.ConfChange)
```



```
        (<-chan *string, <-chan error, <-chan *snap.Snapshotter) {
    // 创建 commitC 和 errorC 通道，前面已经介绍过其功能，不再赘述
    commitC := make(chan *string)
    errorC := make(chan error)

    rc := &raftNode{
        proposeC:    proposeC,
        confChangeC: confChangeC,
        commitC:     commitC,
        errorC:      errorC,
        id:          id,
        peers:       peers,
        join:        join,
        // 初始化存放 WAL 日志和 SnapShot 文件的的目录
        waldir:      fmt.Sprintf("raftexample-%d", id),
        snapdir:     fmt.Sprintf("raftexample-%d-snap", id),
        getSnapshot: getSnapshot,
        snapCount:   defaultSnapCount, // 默认值是 10000
        // 创建 stopc、httpstopc、httpdonec 和 snapshotterReady 四个通道
        stopc:      make(chan struct{}),
        httpstopc:   make(chan struct{}),
        httpdonec:   make(chan struct{}),
        snapshotterReady: make(chan *snap.Snapshotter, 1),
        // 其余字段在 WAL 日志回放完成之后才会初始化
    }

    // 单独启动一个 goroutine 执行 startRaft()方法，在该方法中完成剩余初始化操作
    go rc.startRaft()
    // 将 commitC、errorC 和 snapshotterReady 三个通道返回给上层应用
    return commitC, errorC, rc.snapshotterReady
}
```

下面继续分析 startRaft()函数，其核心步骤如下：

（1）创建 Snapshotter，并将该 Snapshotter 实例返回给上层模块。

（2）创建 WAL 实例，然后加载快照并回放 WAL 日志。

（3）创建 raft.Config 实例，其中包含了启动 etcd-raft 模块的所有配置，Config 中封装的信息在前面已经详细介绍过了，这里不再重复介绍。



（4）初始化底层 etcd-raft 模块，得到前面介绍的 node 实例。

（5）创建 Transport 实例，该实例负责集群中各个节点之间的网络通信，其具体实现在 rafthttp 包中。

（6）建立与集群中其他节点的网络连接。

（7）启动网络组件，其中会监听当前节点与集群中其他节点之间的网络连接，并进行节点之间的消息读写。

（8）启动两个后台 goroutine，它们的主要工作都是处理上层模块与底层 etcd-raft 模块的交互，但处理的具体内容不同，后面会详细介绍这两个 goroutine 的处理流程。

raftNode.startRaft()方法的具体实现如下：

```
func (rc *raftNode) startRaft() {
  // 检测 snapdir 字段指定的目录是否存在，该目录用于存放定期生成的快照数据；
  // 若 snapdir 目录不存在，则进行创建；若创建失败，则输出异常日志并终止程序(略)

  // 步骤 1、创建 Snapshotter 实例，并该 Snapshotter 实例通过 snapshotterReady 通道返回给上层
  // 应用 SnapShotter 实例提供了读写快照文件的功能，后面会详细介绍 SnapShotter 的实现
  rc.snapshotter = snap.New(rc.snapdir)
  rc.snapshotterReady <- rc.snapshotter

  // 步骤 2、创建 WAL 实例，然后加载快照并回放 WAL 日志
  oldwal := wal.Exist(rc.waldir) // 检测 waldir 目录下是否存在旧的 WAL 日志文件
  rc.wal = rc.replayWAL() // 在 replayWAL()方法中会先加载快照数据，然后重放 WAL 日志文件

  // 步骤 3、创建 raft.Config 实例
  rpeers := make([]raft.Peer, len(rc.peers))
  for i := range rpeers {
    rpeers[i] = raft.Peer{ID: uint64(i + 1)}
  }
  c := &raft.Config{
    ID:              uint64(rc.id),
    ElectionTick:    10,           // 对应前面介绍的 election timeout 时间
    HeartbeatTick:   1,            // 心跳计时器的超时时间
    // 持久化存储，与 etcd-raft 模块中的 raftLog.storage 共享同一个 MemoryStorage 实例
    Storage:         rc.raftStorage,
    MaxSizePerMsg:   1024 * 1024, // 每条消息的最大长度
    MaxInflightMsgs: 256, // 已发送但是收到未响应的消息上限个数
```



```
    }

    // 步骤 4、初始化底层的 etcd-raft 模块，这里会根据 WAL 日志的回放情况，判断当前节点是首次启动
    // 还是重新启动，StartNode()和 RestartNode()两个函数在前面已经详细介绍过了，这里不再赘述
    if oldwal {
        rc.node = raft.RestartNode(c) // 重启节点
    } else {
        startPeers := rpeers
        if rc.join {
            startPeers = nil
        }
        rc.node = raft.StartNode(c, startPeers) // 初次启动节点
    }

    // 步骤 5、创建 Transport 实例并启动，它负责 raft 节点之间通信的网络服务，后面详细介绍
    rc.transport = &rafthttp.Transport{
        ID:          types.ID(rc.id),
        ClusterID:   0x1000,
        Raft:        rc,
        ServerStats: stats.NewServerStats("", ""),
        LeaderStats: stats.NewLeaderStats(strconv.Itoa(rc.id)),
        ErrorC:      make(chan error),
    }
    rc.transport.Start() // 启动网络服务相关的组件

    // 步骤 6、建立与集群中其他各个节点的连接，后面详细介绍其实现
    for i := range rc.peers {
        if i+1 != rc.id {
            rc.transport.AddPeer(types.ID(i+1), []string{rc.peers[i]})
        }
    }

    // 步骤 7、启动一个 goroutine，其中会监听当前节点与集群中其他节点之间的网络连接，下面会详细介绍
    go rc.serveRaft()

    // 步骤 8、启动后台 goroutine 处理上层应用与底层 etcd-raft 模块的交互
    go rc.serveChannels()
}
```



介绍完 startRaft()函数的大致流程之后，我们着重分析其中几个关键步骤的具体实现。raftNode.replayWAL()方法首先会读取快照数据，在快照的元数据中记录了该快照包含的最后一条 Entry 记录的 Term 值和索引值。然后根据该 Term 值和索引值确定读取 WAL 日志文件的位置，并进行日志记录的读取。最后将读取到的快照数据、WAL 日志记录和状态信息保存到 raftNode.raftStorage（即 etcd-raft 模块中 raftLog.storage）中。raftNode.replayWAL()方法的具体实现如下：

```
func (rc *raftNode) replayWAL() *wal.WAL {
    // 读取快照文件，该方法底层会调用 Snapshotter.Load()方法完成快照文件的读取，Snapshotter 的
    // 具体实现后面详细介绍
    snapshot := rc.loadSnapshot()
    w := rc.openWAL(snapshot) // 根据读取到的 SnapShot 实例的元数据创建 WAL 实例
    _, st, ents, err := w.ReadAll() // 读取快照数据之后的全部 WAL 日志数据，并获取状态信息

    // 异常检查，若读取 WAL 日志文件的过程中出现异常，则输出日志并终止程序（略）

    // 创建 MemoryStorage 实例，MemoryStorage 的主要功能在前面已经介绍过了，这里不再赘述
     rc.raftStorage = raft.NewMemoryStorage()
    if snapshot != nil {
        rc.raftStorage.ApplySnapshot(*snapshot) // 将快照数据加载到 MemoryStorage 中
    }
    // 将读取 WAL 日志之后得到的 HardState 加载到 MemoryStorage 中
    rc.raftStorage.SetHardState(st)
    // 将读取 WAL 日志得到的 Entry 记录加载到 MemoryStorage 中
    rc.raftStorage.Append(ents)

    if len(ents) > 0 { // 快照之后存在已经持久化的 Entry 记录，这些记录需要回放到上层应用的状态机中
        rc.lastIndex = ents[len(ents)-1].Index // 更新 raftNode.lastIndex，记录了回放的结
                                               // 束位置
    } else {
        // 快照之后不存在已持久化的 Entry 记录，则向 commitC 中写入 nil，在下面还会看到，
        // 当 WAL 日志全部回放完成之后，也会向 commitC 通道中写入 nil 作为信号
        rc.commitC <- nil
    }
    return w
}
```



仅供非商业用途或交流学习使用

WAL 对日志文件的管理在后面会有单独的章节详细介绍，现在读者只要了解 WAL 实例的主要功能是读写日志文件即可，这里只是简单看一下 raftNode.openWAL()方法中确定读取 WAL 日志文件的位置及创建 WAL 实例的过程：

```
func (rc *raftNode) openWAL(snapshot *raftpb.Snapshot) *wal.WAL {
   if !wal.Exist(rc.waldir) { // 检测 WAL 日志目录是否存在，如果不存在则进行创建
      if err := os.Mkdir(rc.waldir, 0750);... // 异常处理(略)
      // 新建 WAL 实例，其中会创建相应目录和一个空的 WAL 日志文件
      w, err := wal.Create(rc.waldir, nil)
      w.Close() // 关闭 WAL，其中包括各种关闭目录、文件和相关的 goroutine
   }

   // 创建 walsnap.Snapshot 实例并初始化其 Index 字段和 Term 字段，注意两者的区别：walsnap.Snapshot
   // 只包含了快照元数据中的 Term 值和索引值，并不会包含真正的快照数据
   walsnap := walpb.Snapshot{}
   if snapshot != nil {
      walsnap.Index, walsnap.Term = snapshot.Metadata.Index, snapshot.Metadata.Term
   }
   w, err := wal.Open(rc.waldir, walsnap) // 创建 WAL 实例，异常处理(略)
   return w
}
```

完成 WAL 日志回放之后，会创建 Config 对象并进行 etcd-raft 模块的初始化，这个过程在前面介绍 node 结构体时已经介绍过，这里不再展开详述。之后就是 rafthttp.Transport 的初始化过程，在后面有单独的章节介绍 etcd-rafthttp 模块，读者这里只需要了解 rafthttp.Transport 主要负责管理当前节点与集群其他节点的网络连接即可。

在 raftNode.startRaft()方法的最后，启动了两个后台 goroutine 分别执行了 serveRaft()方法和 serveChannels()方法，其中 serveRaft()方法负责监听当前节点的地址，完成与其他节点的通信。下面先来看一下 serveRaft()方法的具体实现：

```
func (rc *raftNode) serveRaft() {
   url, err := url.Parse(rc.peers[rc.id-1]) // 获取当前节点的 URL 地址
   // 异常检查，若解析 URL 地址过程出现异常，则输出日志并终止程序(略)

   // 创建 stoppableListener 实例，stoppableListener 继承了 net.TCPListener(当然也实
   // 现了 net.Listener 接口)接口，它会与 http.Server 配合实现对当前节点的 URL 地址进行监听
   ln, err := newStoppableListener(url.Host, rc.httpstopc)
```



仅供非商业用途或交流学习使用

```
    // 异常检查(略)

    // 创建 http.Server 实例，它会通过上面的 stoppableListener 实例监听当前节点的 URL 地址
    // stoppableListener.Accept()方法监听到新连接到来时，会创建对应的 net.Conn 实例，http.Server
    // 会为每个连接创建单独的 goroutine 处理，每个请求都会由 http.Server.Handler 处理。这里的
    // Handler 是由 rafthttp.Transporter 创建的，后面详细介绍 rafthttp.Transporter 的具体实
    // 现。另外需要读者了解的是，http.Server.Serve()方法会一直阻塞，直到 http.Server 关闭
    err = (&http.Server{Handler: rc.transport.Handler()}).Serve(ln)

    select {
    case <-rc.httpstopc:
    default: // 日志输出(略)
    }
    close(rc.httpdonec)
}
}
```

下面再来看一下 serveChannels()方法，其中会单独启动一个后台 goroutine 来负责上层模块传递给 etcd-raft 模块的数据，主要处理前面介绍的 proposeC、confChangeC 两个通道。serveChannels()方法的具体实现如下：

```
func (rc *raftNode) serveChannels() {
    // 前面介绍的 raftNode.replayWAL()方法读取了快照数据、WAL 日志等信息，并记录到了
    // raftNode.raftStorage 中，这里就会使用这些数据

    snap, err := rc.raftStorage.Snapshot() // 获取快照数据和快照的元数据，异常处理(略)
    rc.confState = snap.Metadata.ConfState
    rc.snapshotIndex = snap.Metadata.Index
    rc.appliedIndex = snap.Metadata.Index

    // 创建一个每隔 100ms 触发一次的定时器，那么在逻辑上，100ms 即是 etcd-raft 组件的最小时间单位，
    // 该定时器每触发一次，则逻辑时钟推进一次
    ticker := time.NewTicker(100 * time.Millisecond)

    // 单独启动一个 goroutine 负责将 proposeC、confChangeC 通道上接收到的数据传递给 etcd-raft
    // 组件进行处理
    go func() {
        var confChangeCount uint64 = 0
```



仅供非商业用途或交流学习使用

```
        for rc.proposeC != nil && rc.confChangeC != nil {
            select {
            case prop, ok := <-rc.proposeC: // 收到上层应用通过 proposeC 通道传递过来的数据
                // 如果发生异常，则将 raftNode.proposeC 字段置空，当前循环及整个 goroutine 都会结束
                // 通过 node.Propose()方法，将数据传入底层 etcd-raft 组件进行处理
                rc.node.Propose(context.TODO(), []byte(prop))

            case cc, ok := <-rc.confChangeC: // 收到上层应用通过 confChangeC 通道传递过来的数据
                // 如果发生异常，则将 raftNode.confChangeC 字段置空(略)
                confChangeCount += 1 // 统计集群变更请求的个数，并将其作为 ID
                cc.ID = confChangeCount
                // 通过 node.ProposeConfChange()方法，将数据传入底层 etcd-raft 组件进行处理，
                rc.node.ProposeConfChange(context.TODO(), cc)
            }
        }
        close(rc.stopc) // 关闭 stopc 通道，触发 rafeNode.stop()方法的调用
    }()
    // 剩余的代码下面会详细描述

}
```

另外，serveChannel()方法还负责处理 etcd-raft 模块返回给上层模块的数据及其他相关的操作，例如，推进 etcd-raft 模块的逻辑时钟、处理网络异常等，相关代码片段如下所示。

```
func (rc *raftNode) serveChannels() {
    // 前面对 proposeC 和 confChangeC 两个通道的处理已经介绍过了，相关代码这里不再粘贴了

    for { // 该循环主要负责处理底层 etcd-raft 组件返回的 Ready 数据
        select {
        case <-ticker.C:
            rc.node.Tick() // 上述 ticker 定时器触发一次，即会推进 etcd-raft 组件的逻辑时钟

        // 读取 node.readyc 通道，前面介绍 etcd-raft 组件时也提到，该通道是 etcd-raft 组件与上层应用
        // 交互的主要通道之一，其中传递的 Ready 实例也封装了很多信息，读者可以回顾前面对 node 的介绍
        case rd := <-rc.node.Ready():
            // 将当前 etcd-raft 组件的状态信息，以及待持久化的 Entry 记录先记录到 WAL 日志文件中，
            // 即使之后宕机，这些信息也可以在节点下次启动时，通过前面回放 WAL 日志的方式进行恢复
            // WAL 记录日志的具体实现，在后面的章节中做详细介绍，这里不做过多描述
```



仅供非商业用途或交流学习使用

```
            rc.wal.Save(rd.HardState, rd.Entries)

            if !raft.IsEmptySnap(rd.Snapshot) {  // 检测 etcd-raft 组件生成了新的快照数据
                rc.saveSnap(rd.Snapshot) // 将新的快照数据写入快照文件中
                // 将新快照持久化到 raftStorage，MemoryStorage 的实现在后面的章节详细介绍
                rc.raftStorage.ApplySnapshot(rd.Snapshot)
                rc.publishSnapshot(rd.Snapshot) // 通知上层应用加载新快照
            }
            // 将待持久化的 Entry 记录追加到 raftStorage 中完成持久化
            rc.raftStorage.Append(rd.Entries)
            // 将待发送的消息发送到指定节点，Transport 的具体实现在后面的章节中做介绍，这里不做过多描述
            rc.transport.Send(rd.Messages)

            // 将已提交、待应用的 Entry 记录应用到上层应用的状态机中，异常处理(略)
            if ok := rc.publishEntries(rc.entriesToApply(rd.CommittedEntries)); ...

            // 随着节点的运行，WAL 日志量和 raftLog.storage 中的 Entry 记录会不断增加，
            // 所以节点每处理 10000 条（默认值）Entry 记录，就会触发一次创建快照的过程，
            // 同时 WAL 会释放一些日志文件的句柄，raftLog.storage 也会压缩其保存的 Entry 记录
            rc.maybeTriggerSnapshot()

            // 上层应用处理完该 Ready 实例，通知 etcd-raft 组件准备返回下一个 Ready 实例
            rc.node.Advance()
        case err := <-rc.transport.ErrorC: // 处理网络异常
            rc.writeError(err) // 关闭与集群中其他节点的网络连接
            return

        case <-rc.stopc: // 处理关闭命令
            rc.stop()
            return
        }
    }
}
```

接下来，我们详细分析一下处理 Ready 实例中封装的各项数据的过程，首先来看一下 raftNode.saveSnap()方法，其中会将新生成的快照数据保存到磁盘上，还会根据快照的元数据释放部分旧 WAL 日志文件的句柄，其具体实现如下：



仅供非商业用途或交流学习使用

```
func (rc *raftNode) saveSnap(snap raftpb.Snapshot) error {
    // 将新快照数据写入快照文件中，异常处理(略)
    if err := rc.snapshotter.SaveSnap(snap); err != nil {
        return err
    }
    // 根据快照的元数据，创建 walpb.Snapshot 实例
    walSnap := walpb.Snapshot{
        Index: snap.Metadata.Index,
        Term:  snap.Metadata.Term,
    }
    // WAL 会将上述快照的元数据信息封装成一条日志记录下来，WAL 的实现在后面的章节中详细介绍
    if err := rc.wal.SaveSnapshot(walSnap); err != nil {
        return err
    }
    // 根据快照的元数据信息，释放一些无用的 WAL 日志文件的句柄，WAL 的实现在后面的章节中详细介绍
    return rc.wal.ReleaseLockTo(snap.Metadata.Index)
}
```

在 raftNode.publishSnapshot()方法中，会通知上层模块加载新生成的快照数据，并使用新快照的元数据更新 raftNode 中的相应字段，具体实现如下：

```
func (rc *raftNode) publishSnapshot(snapshotToSave raftpb.Snapshot) {
    // 对快照数据进行一系列检测(略)
    // 使用 commitC 通道通知上层应用加载新生成的快照数据
    rc.commitC <- nil // trigger kvstore to load snapshot
    // 记录新快照的元数据
    rc.confState = snapshotToSave.Metadata.ConfState
    rc.snapshotIndex = snapshotToSave.Metadata.Index
    rc.appliedIndex = snapshotToSave.Metadata.Index
}
```

最后来看一下 raftNode 是如何处理待应用的 Entry 记录的，首先是 raftNode.entriesToApply()方法，其中会对 Ready 实例中携带的待应用记录进行过滤，具体实现如下：

```
func (rc *raftNode) entriesToApply(ents []raftpb.Entry) (nents []raftpb.Entry) {
    // 检测 ents 长度(略)
    firstIdx := ents[0].Index
    // 检测 firstIndex 是否合法(略)
```



仅供非商业用途或交流学习使用

```
    // 过滤掉已经被应用过的 Entry 记录
    if rc.appliedIndex-firstIdx+1 < uint64(len(ents)) {
        nents = ents[rc.appliedIndex-firstIdx+1:]
    }
    return
}
```

其次就是 raftNode.publishEntries()方法，在该方法中，raftNode 会将所有待应用记录写入 commitC 通道中。后续 kvstore 就可以读取 commitC 通道并保存相应的键值对数据，其具体实现如下：

```
func (rc *raftNode) publishEntries(ents []raftpb.Entry) bool {
    for i := range ents {
        switch ents[i].Type {
        case raftpb.EntryNormal:
            // 如果 Entry 记录的 Data 为空，则直接忽略该条 Entry 记录(略)
            s := string(ents[i].Data)
            select { // 将数据写入 commitC 通道，kvstore 会读取从其中读取并记录相应的 KV 值
            case rc.commitC <- &s:
            }

        case raftpb.EntryConfChange:
            var cc raftpb.ConfChange // 将 EntryConfChange 类型的记录封装成 ConfChange
            cc.Unmarshal(ents[i].Data)
            // 将 ConfChange 实例传入底层的 etcd-raft 组件，其中的处理过程在前面已经详细分析过了
            rc.confState = *rc.node.ApplyConfChange(cc)

            // 除了 etcd-raft 组件中需要创建(或删除)对应的 Progress 实例，网络层也需要做出相应的调
            // 整，即添加(或删除)相应的 Peer 实例
            switch cc.Type {
            case raftpb.ConfChangeAddNode:
                if len(cc.Context) > 0 {
                    rc.transport.AddPeer(types.ID(cc.NodeID), []string{string(cc.Context)})
                }
            case raftpb.ConfChangeRemoveNode:
                if cc.NodeID == uint64(rc.id) {
                    log.Println("I've been removed from the cluster! Shutting down.")
                    return false
```



仅供非商业用途或交流学习使用

```
            }
            rc.transport.RemovePeer(types.ID(cc.NodeID))
        }
    }
    // 处理完成之后，更新 raftNode 记录的已应用位置，该值在过滤已应用记录的 entriesToApply()
    // 方法及后面即将介绍的 maybeTriggerSnapshot()方法中都有使用
    rc.appliedIndex = ents[i].Index

    // 此次应用的是否为重放的 Entry 记录，如果是，且重放完成，则使用 commitC 通道通知 kvstore
    if ents[i].Index == rc.lastIndex {
        select {
        case rc.commitC <- nil:
        }
    }
    return true
}
```

为了释放底层 etcd-raft 模块中无用的 Entry 记录，节点每处理指定条数（默认是 10000 条）的记录之后，就会触发一次快照生成操作，相关实现在 raftNode.maybeTriggerSnapshot()方法中，如下所示。

```
func (rc *raftNode) maybeTriggerSnapshot() {
    // 检测处理的记录数是否足够，如果不足，则直接返回(略)

    // 获取快照数据，在 raftexample 示例中是获取 kvstore 中记录的全部键值对数据，异常处理(略)
    data, err := rc.getSnapshot()
    // 创建 Snapshot 实例，同时也会将快照和元数据更新到 raftLog.MemoryStorage 中
    snap, err := rc.raftStorage.CreateSnapshot(rc.appliedIndex,
        &rc.confState, data)

    // 保存快照数据，raftNode.saveSnap()方法在前面已经介绍过了，这里不再赘述，异常处理(略)
    if err := rc.saveSnap(snap);...

    compactIndex := uint64(1)  // 计算压缩的位置，压缩之后，该位置之前的全部记录都会被抛弃
    if rc.appliedIndex > snapshotCatchUpEntriesN {
        compactIndex = rc.appliedIndex - snapshotCatchUpEntriesN
    }
```



仅供非商业用途或交流学习使用

```
    // 压缩 raftLog 中保存的 Entry 记录，MemoryStorage.Compact()方法在前介绍过，这里不再赘述
    if err := rc.raftStorage.Compact(compactIndex); ... // 异常处理(略)
    rc.snapshotIndex = rc.appliedIndex // 更新 raftNode 的相关字段
}
```

到此为止，raftexample 示例中最核心的内容已经介绍完了，在本节后续，将简单介绍该示例中其他相关功能组件。

3.3.2　HTTP服务端

在 raftexample 示例中，会初始化一个简单的 HTTP 服务端来接收用户发送的 HTTP 请求，也是 raftexample 示例与用户交互的唯一途径，本小节就将介绍该服务端的具体实现。在该 HTTP 服务中只有一个结构体，即 httpKVAPI，其核心字段如下。

- store（*kvstore 类型）：kvstore 实例在 raftexample 示例中扮演了持久化存储的角色，用于保存用户提交的键值对信息。
- confChangeC（chan<- raftpb.ConfChange 类型）：在 raftexample 示例中，当用户发送 POST （或 DELETE）请求时，会被认为是发送了一个集群节点增加（或删除）的请求，httpKVAPI 会将该请求的信息写入 confChangeC 通道，正如上一小节所介绍的，raftNode 实例会读取 confChange 通道并进行相应处理。

httpKVAPI 实现了 http.Handler 接口，在 httpKVAPI.ServeHTTP()方法中，会根据 HTTP 请求的 Method 进行分类处理，其中只处理 PUT、GET、POST 和 DELETE 四种 Method，相关的代码片段如下所示。

- PUT 请求表示向 kvstore 实例中添加（或更新）指定的键值对数据。

```
func (h *httpKVAPI) ServeHTTP(w http.ResponseWriter, r *http.Request) {
    key := r.RequestURI // 获取请求的 URI 作为 key
    switch { // 省略处理其他 HTTP Method 的相关代码
    case r.Method == "PUT":                    // PUT 请求的处理
        v, err := ioutil.ReadAll(r.Body) // 读取 HTTP 请求体，读取过程的异常处理(略)

        // 在 kvstore.Propose()方法中会对键值对进行序列化，之后将结果写入 proposeC 通道，
        // 后续 raftNode 会读取其中数据进行处理，raftNode 在前面已经详细介绍过了，这里不再赘述
        h.store.Propose(key, string(v))
        w.WriteHeader(http.StatusNoContent) // 向用户返回相应的状态码
    }
}
```



仅供非商业用途或交流学习使用

- GET 请求表示从 kvstore 实例中读取指定的键值对数据。

```
func (h *httpKVAPI) ServeHTTP(w http.ResponseWriter, r *http.Request) {
    key := r.RequestURI // 获取请求的 URI 作为 key
    switch { // 省略处理其他 HTTP Method 的相关代码
    case r.Method == "GET": // GET 请求的处理
        // 直接从 kvstore 中读取指定的键值对数据，并返回给用户
        if v, ok := h.store.Lookup(key); ok {
            w.Write([]byte(v))
        } else {
            http.Error(w, "Failed to GET", http.StatusNotFound) // 返回 404
        }
    }
}
```

- POST 请求表示向集群中新增指定的节点。

```
func (h *httpKVAPI) ServeHTTP(w http.ResponseWriter, r *http.Request) {
    key := r.RequestURI // 获取请求的 URI 作为 key
    switch { // 省略处理其他 HTTP Method 的相关代码
    case r.Method == "POST":   // POST 请求的处理
        url, err := ioutil.ReadAll(r.Body) // 读取请求体，获取新加节点的 URL，异常处理(略)
        // 解析 URI 得到新增节点的 id，异常处理(略)
        nodeId, err := strconv.ParseUint(key[1:], 0, 64)

        cc := raftpb.ConfChange{// 创建 ConfChange 消息
            Type: raftpb.ConfChangeAddNode, // ConfChangeAddNode 即表示新增节点
            NodeID: nodeId, // 指定新增节点的 ID
            Context: url,   // 指定新增节点的 URL
        }
        h.confChangeC <- cc // 将 ConfChange 实例写入 confChangeC 通道中
        w.WriteHeader(http.StatusNoContent) // 返回相应状态码
    }
}
```

- DELETE 请求表示从集群中删除指定的节点。

```
func (h *httpKVAPI) ServeHTTP(w http.ResponseWriter, r *http.Request) {
```



```
    key := r.RequestURI // 获取请求的 URI 作为 key
    switch { // 省略处理其他 HTTP Method 的相关代码
    case r.Method == "DELETE": // DELETE 请求的处理
        // 解析 key 得到的待删除的节点 id，异常处理(略)
        nodeId, err := strconv.ParseUint(key[1:], 0, 64)

        cc := raftpb.ConfChange{
            Type:   raftpb.ConfChangeRemoveNode, // ConfChangeRemoveNode 表示删除指定节点
            NodeID: nodeId, // 指定待删除的节点 id
        }
        h.confChangeC <- cc // 将 ConfChange 实例发送到 confChangeC 通道中
        // 与 PUT 请求的处理方式类似，向客户端返回状态码为 204 的 HTTP 响应
        w.WriteHeader(http.StatusNoContent)
    }
}
```

在 raftexample 完成初始化之后，会调用 serveHttpKVAPI()函数，其中会启动一个单独的后台 goroutine 来监听指定的地址，接收用户发来的 HTTP 请求。serveHttpKVAPI()函数的具体实现如下：

```
func serveHttpKVAPI(kv *kvstore, port int, confChangeC chan<- raftpb.ConfChange,
        errorC <-chan error) {
    srv := http.Server{ // 创建 http.Server 用于接收 HTTP 请求
        Addr: ":" + strconv.Itoa(port),
        Handler: &httpKVAPI{ // 设置 http.Server 的 Handler 字段，即上述 httpKVAPI 实例
            store:      kv,
            confChangeC: confChangeC,
        },
    }

    // 启动单独的 goroutine 来监听 Addr 指定的地址，当有 HTTP 请求时，http.Server 会创建对应的
    // goroutine，并调用前面介绍的 httpKVAPI.ServeHTTP()方法进行处理
    go func() {
        if err := srv.ListenAndServe();... // 异常处理(略)
    }()
}
```



3.3.3 kvstore

在 raftexample 示例中，kvstore 扮演了持久化存储和状态机的角色，etcd-raft 模块通过 Ready 实例返回的待应用 Entry 记录最终都会存储到 kvstore 中。kvstore 中的核心字段如下。

- proposeC (chan<- string 类型)：在前面的章节中提到，httpKVAPI 处理 HTTP PUT 请求时，会调用 kvstore.Propose()方法将用户请求的数据写入 proposeC 通道中，之后 raftNode 会从该通道中读取数据并进行处理。
- kvStore (map[string]string 类型)：该字段是用来存储键值对的 map，其中存储的键值都是 string 类型的。
- snapshotter (*snap.Snapshotter 类型)：该字段负责读取快照文件。

通过前面对 raftNode 的介绍可知，它会将待应用的 Entry 记录写入 commitC 通道中，另外，当需要加载快照数据时，它会向 commitC 通道写入 nil 作为信号。在 kvstore 初始化（newKVStore() 函数）时，会启动一个后台 goroutine 来执行 kvstore.readCommits()，其中就会读取 commitC 通道，然后将读取到的键值对写入 kvStore 中，其具体实现如下：

```
func (s *kvstore) readCommits(commitC <-chan *string, errorC <-chan error) {
  for data := range commitC { // 循环读取 commitC 通道
    if data == nil { // 读取到 nil 时，则表示需要读取快照数据
      // 通过 snapshotter 读取快照文件，省略加载异常的处理，snapshotter 的具体实现后面详细分析
      snapshot, err := s.snapshotter.Load()
      if err := s.recoverFromSnapshot(snapshot.Data); ... // 加载快照数据，省略异常处理
      continue
    }

    var dataKv kv // 将读取到的数据进行反序列化得到 kv 实例
    dec := gob.NewDecoder(bytes.NewBufferString(*data))
    s.mu.Lock() // 加锁
    s.kvStore[dataKv.Key] = dataKv.Val // 将读取到的键值对保存到 kvStore 中
    s.mu.Unlock()// 解锁
  }
}
```

在 kvstore.recoverFromSnapshot()方法中，会将快照数据反序列化得到一个 map 实例，然后直接替换当前的 kvStore 实例，具体实现如下：

```
func (s *kvstore) recoverFromSnapshot(snapshot []byte) error {
```



```
    var store map[string]string
    if err := json.Unmarshal(snapshot, &store); ... // 快照数据反序列化
    s.mu.Lock() // 加锁
    s.kvStore = store // 替换 kvStore 字段
    s.mu.Unlock() // 解锁
    return nil
}
```

在 httpKVAPI 处理 HTTP GET 请求时，会直接调用 kvstore.Lookup()方法读取 kvStore 中记录的键值对，其实现比较简单，这里不再展开介绍，感兴趣的读者可以参考源码进行学习。

最后简单说一下，在 raftexample 示例 main()方法中，会根据配置创建 kvstore 实例、raftNode 实例、简易的 HTTP 服务端及上述各种通道，并将这些组件组装起来，其具体实现比较简单，代码就不再粘贴出来了，感兴趣的读者可以参考相关代码进行学习。

本章小结

本章首先介绍了 etcd-raft 模块对 Raft 协议中相关概念的抽象及其自身实现使用到的基本数据结构，例如，Config、MemoryStorage、unstable、raftLog 等。然后详细介绍了 raft 结构体的实现，它是 etcd-raft 模块的核心，其中处理了所有 Raft 协议相关的各种类型的消息，同时也负责节点的状态切换。之后，又介绍了 Node 接口及其 etcd-raft 模块中提供的实现（结构体 node），它是对底层 raft 实例的一层封装，对外提供了更简单、更易用的 API。

在本章的最后一节中，我们分析了 etcd 自带的 raftexample 示例，其中涉及该示例的大致架构、具体实现，以及使用底层 etcd-raft 模块的方式。希望读者通过本章的阅读，可以加深对 Raft 协议的理解，了解 etcd-raft 模块的实现原理，并且能够类似 raftexample 示例那样使用 etcd-raft 模块，对外提供基本 Raft 协议的功能。





第 4 章 网络层

4.1 Go语言网络编程基础

4.1.1 http.Server

在开始介绍 etcd 中提供的网络模块之前，需要读者先了解一下 Go 语言的网络编程，特别是http.Server等组件的使用方法和原理，这里要介绍的就是我们在实际开发中最常用的http库。首先来看其中的几个概念。

- Request：表示客户端请求，负责解析 HTTP 请求信息。
- Response：表示服务端返回给客户端的响应。
- Conn：表示一次网络链接。
- Handle：其中封装了处理客户端的请求的逻辑及创建返回响应的逻辑。

在前面介绍的raftexample示例中，raftNode.serveRaft()方法中有一段使用的HTTP库的代码，大致实现如下：

```
func (rc *raftNode) serveRaft() {
    url, err := url.Parse(rc.peers[rc.id-1]) // 获取当前节点的 URL 地址
    // 若解析 URL 地址过程出现异常，则输出日志并终止程序(略)

    // 创建 stoppableListener 实例，stoppableListener 内嵌了 net.TCPListener(当然也就是实
```



```
    // 现了 net.Listener 接口)接口，它会与 http.Server 配合实现对当前节点的 URL 地址进行监听
    ln, err := newStoppableListener(url.Host, rc.httpstopc)
    // 异常检查(略)

    // 创建 http.Server 实例
    err = (&http.Server{Handler: rc.transport.Handler()}).Serve(ln)
}
```

这里创建了一个 http.Server 实例，它会通过传入的 TCPListener 实例监听指定地址上的连接请求，当 TCPListener 实例通过 Accept()方法监听到新连接到来时，会创建对应的 net.Conn 实例（读者可以将其理解为一个单独的网络连接），http.Server 会为每个连接启动一个独立的后台 goroutine，处理该连接上的请求。每个请求的处理逻辑封装在 http.Server.Handler 中（创建 http.Server 实例时传入），大致原理如图 4-1 所示。

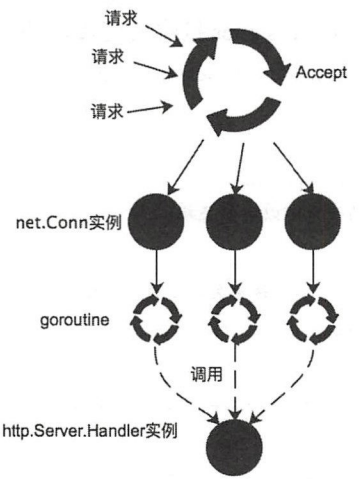


图 4-1

在 raftexample 示例中使用的 Handler 实例是由 rafthttp.Transporter.Handler()方法创建的，其中为多个路径绑定了对应的 Handler 实例。这里简单看一下该方法的具体实现：

```
func (t *Transport) Handler() http.Handler {
    // 创建 pipelineHandler、streamHandler 和 snapshotHandler 三个实例，这三个实例都实现
    // 了 http.Server.Handler 接口，具体实现在本章后面介绍 rafthttp.Transport 时详细分析
    pipelineHandler := newPipelineHandler(t, t.Raft, t.ClusterID)
    streamHandler := newStreamHandler(t, t, t.Raft, t.ID, t.ClusterID)
    snapHandler := newSnapshotHandler(t, t.Raft, t.Snapshotter, t.ClusterID)
```



```
    // 创建 ServeMux 实例，ServeMux 是一个多路复用器，ServeMux 主要通过其 m 字段
    //（map[string]muxEntry 类型）存储具体的 URL 和 Handler 实例之间的映射关系，
    // 后面还会详细介绍 ServeMux.Handle() 方法的实现
    mux := http.NewServeMux()
    // 下面就是设置 URL 与 Handler 实例的映射关系，也就是访问指定 URL 地址的请求，由对应的 Handler
    // 实例进行处理
    mux.Handle(RaftPrefix, pipelineHandler)         // "/raft"--> pipelineHandler
    mux.Handle(RaftStreamPrefix+"/", streamHandler) // "/raft/stream/"--> streamHandler
    mux.Handle(RaftSnapshotPrefix, snapHandler)     // "/raft/snapshot"--> snapHandler
    mux.Handle(ProbingPrefix, probing.NewHandler()) // "/raft/probing"--> httpHealth
    return mux
}
```

下面我们深入介绍一下 http.Server 是如何与 net.TCPListener 实例配合工作的。首先是 http.Server. Serve()方法，其中通过 net.Listener 接收新连接，并为每个连接启动一个单独的后台 goroutine 来处理该连接上的请求。http.Server. Serve()方法的代码比较长，笔者将其进行了精简，其核心代码如下：

```
func (srv *Server) Serve(l net.Listener) error {
    // 省略一些其他代码，感兴趣的读者可以查阅源码
    for {
        // 接收新连接(net.Conn)，下面紧接着会以 raftexample 中的 stoppableListener 为例，
        // 分析 Accept() 方法的实现
        rw, e := l.Accept()
        // 异常处理，会导致当前循环结束，Serve() 方法返回(略)
        tempDelay = 0
        c := srv.newConn(rw) // 将 net.Conn 封装成 http.conn
        c.setState(c.rwc, StateNew) // before Serve can return
        go c.serve(ctx) // 启动单独的 goroutine 处理新连接
    }
}
```

这里以 raftexample 示例中的 stoppableListener 为例来分析 net.Listener 的工作原理，stoppableListener 内嵌了 net.TCPListener 接口，其定义如下：

```
type stoppableListener struct {
    *net.TCPListener // 内嵌 net.TCPListener
    // stopc 通道用来监听是否应该关闭当前的 Server，也就是 raftNode 中的 httpstopc 通道
```



```
    stopc <-chan struct{}
}
```

stoppableListener.Accept()方法重写了 TCPListener.Accept()方法，具体实现如下：

```
func (ln stoppableListener) Accept() (c net.Conn, err error) {
    connc := make(chan *net.TCPConn, 1)
    errc := make(chan error, 1)
    go func() { // 启动单独的 goroutine 来接收新连接
        tc, err := ln.AcceptTCP()// 异常处理(略)
        connc <- tc // 将接收到新连接传递到 connc 通道中
    }()
    select {
    case <-ln.stopc: // 从 stopc 通道中接收到通知时，会抛出异常
        return nil, errors.New("server stopped")
    case err := <-errc: // 如果在接收新连接时出现异常，则抛出异常
        return nil, err
    case tc := <-connc: // 接收到新建连接，设置 KeepAlive 等连接属性，并返回
        tc.SetKeepAlive(true)
        tc.SetKeepAlivePeriod(3 * time.Minute)
        return tc, nil
    }
}
```

回到 http.Server.Serve()方法继续分析，我们进入 http.conn.serve()方法，看看它是如何处理单个连接及其上请求的，其具体实现如下：

```
func (c *conn) serve(ctx context.Context) {
    c.remoteAddr = c.rwc.RemoteAddr().String()
    ctx = context.WithValue(ctx, LocalAddrContextKey, c.rwc.LocalAddr())
    defer func() { ... }() // 连接关闭相关的操作(略)
    // 省略一些 TLS 相关的代码
    for {
        w, err := c.readRequest(ctx) // 读取客户端的请求，省略异常处理的相关代码
        // 对特殊请求头进行处理(略)
        // 处理请求的其他状态(略)

        // 创建 serverHandler 实例，并调用其 ServeHTTP()方法处理请求，需要注意的是，
```



```
        // 这里 serverHandler.srv 字段指向的就是前面创建的 http.Server 实例
        serverHandler{c.server}.ServeHTTP(w, w.req)

        // 处理完请求之后，就会根据 KeepAlives、idleTimeout 等配置决定是否断开连接(略)
    }
}
```

处理单个请求的核心就是 serverHandler.ServeHTTP()方法，该方法会将请求交给绑定的 http.Server.Handler 实例进行处理，具体实现如下：

```
func (sh serverHandler) ServeHTTP(rw ResponseWriter, req *Request) {
    // http.Server.Handler 字段指向的就是前面通过 Transport.Handler()方法创建的
    // http.ServeMux 实例
    handler := sh.srv.Handler
    if handler == nil { // 如果 http.Server 未初始化 Handler 字段，则使用默认的 ServeMux 实例
        handler = DefaultServeMux
    }
    // 特殊的请求路径或请求方法，会选择其他 Handler 进行处理(略)
    handler.ServeHTTP(rw, req) // 调用 http.ServeMux.ServeHTTP()方法进行处理
}
```

http.ServeMux.ServHttp()方法会根据请求的具体路径，选择合适的 Handler 实例处理请求，该方法的具体实现如下：

```
func (mux *ServeMux) ServeHTTP(w ResponseWriter, r *Request) {
    // 特殊请求路径的处理(略)
    h, _ := mux.Handler(r) // 根据请求路径选择对应的 Handler
    h.ServeHTTP(w, r) // 将请求交给 Handler 进行处理
}

// 下面是 ServeMux.Handler()方法的具体实现
func (mux *ServeMux) Handler(r *Request) (h Handler, pattern string) {
    // 对 CONNECT 请求方法的处理(略)
    host := stripHostPort(r.Host) // 从请求中获取 host 信息
    path := cleanPath(r.URL.Path) // 从请求中获取请求路径
    return mux.handler(host, r.URL.Path) // 根据 host 信息和请求路径查找对应的 Handler 实例
}
```



```
// 下面是 ServeMux.handler()方法的具体实现
func (mux *ServeMux) handler(host, path string) (h Handler, pattern string) {
    if mux.hosts {  // 首先根据 host 和 path 的组合查找
        h, pattern = mux.match(host + path)
    }
    if h == nil {  // 只根据 path 查找
        h, pattern = mux.match(path)
    }
    return
}

// 下面是 ServeMux.match()方法的具体实现
func (mux *ServeMux) match(path string) (h Handler, pattern string) {
    v, ok := mux.m[path] // 如果存在精确匹配的 Handler，则直接返回
    if ok { return v.h, v.pattern }

    for k, v := range mux.m { // 查找与 path 匹配程度最高的 Handler 实例
        // 具体查找过程(略)
    }
    return
}
```

http.Server 的大致原理和具体实现就介绍到这里，同时，我们也介绍了 raftexample 示例中使用 http.Server 的一个典型实例，也深入介绍了 http.Server 工作流程。

4.1.2　RoundTripper

在使用 net.http 库发送 HTTP 请求的过程中，最后都调用了 http.Transport.RoundTrip()方法，http.Transport 即 RoundTripper 接口的实现之一。

在 RoundTripper 接口中，定义了 RoundTrip()方法来完成一个 HTTP 事务，那什么是 HTTP 事务呢？简单来说就是：客户端发送一个 HTTP Request 到服务端，服务端处理完请求之后，返回相应的 HTTP Response 的过程。RoundTripper 接口的定义如下：

```
type RoundTripper interface {
    RoundTrip(*Request) (*Response, error)
}
```



结构体 http.Transport 实现了上面的 RoundTripper 接口，这里重点关注 http.Transport 中的几个核心字段。

- idleConn（map[connectMethodKey][]*persistConn 类型）：persistConn 是在普通网络连接（net.Conn）上的一层封装，一般表示的是 TCP 长连接（即在建立连接并传输数据完成之后，不调用 close()方法关闭连接，后续即可复用该 TCP 连接）。http.Transport 中的 idleConn 字段保存了从 connectMethodKey 到 persistConn 的映射，其中的 connectMethodKey 记录了连接的协议、host 等信息。
- idleConnCh（map[connectMethodKey]chan *persistConn）：http.Transport 中的 idleConnCh 字段用来在多个后台 goroutine 之间相互发送 persistConn 实例，这就可以做到 persistConn 的复用，如图 4-2 所示。
- 另一个需要读者了解的字段是 Dial，它是从外部传入的函数指针，该函数负责创建底层的 TCP 连接。

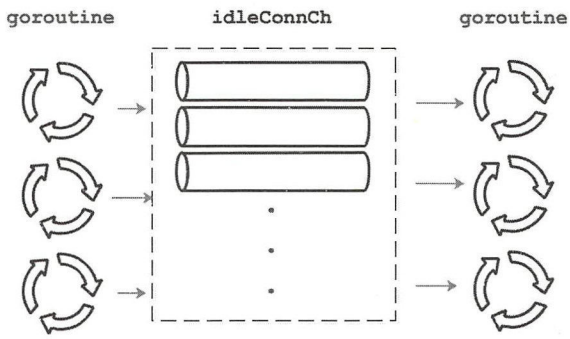


图 4-2

接下来看一下 Transport.RoundTrip()方法的具体实现，它首先会检测 Request 的合法性，然后看一下 altProto 中是否存在请求对应的 RoundTripper 实现。Transport 实际上可以处理不同协议的请求，处理不同协议就要有不同的 RoundTripper 实现，如果需要处理多种协议，就需要通过 RegisterProtocol 来注册一些针对不同协议的 RoundTrip 实现，Transport 发送 Request 之前就是通过该部分逻辑确定到底要使用哪个 RoundTripper 实现的。之后构建一个 connectMethod 实例，通过 Transport.getConn()方法获取 TCP 连接，并调用 persistConn.roundTrip()方法把 Request 写入 TCP 中完成请求的发送。Transport.RoundTrip()方法的具体实现如下：

```
func (t *Transport) RoundTrip(req *Request) (*Response, error) {
    ... ...
    scheme := req.URL.Scheme
    isHTTP := scheme == "http" || scheme == "https"
```



```
    // 对于 HTTP 请求和 HTTPS 进行合法性检查(略)
    altProto, _ := t.altProto.Load().(map[string]RoundTripper)
    if altRT := altProto[scheme]; altRT != nil {
        if resp, err := altRT.RoundTrip(req); err != ErrSkipAltProtocol {
            return resp, err
        }
    }


    for {
        ... ...
        cm, err := t.connectMethodForRequest(treq) // 创建 connectMethod 实例
        pconn, err := t.getConn(treq, cm) // 获取 TCP 连接
        var resp *Response
        resp, err = pconn.roundTrip(treq) // 发送请求，并获取响应
        if err == nil {
            return resp, nil
        }
        ... ... // 如果发送失败，会进行重试
    }
}
```

下面我们继续深入 Transport.getConn()方法，看一下获取 TCP 长连接的过程。在Transport.getConn()方法中，首先会调用 Transport.getIdleConn()方法获取可用的空闲连接，如果没有空闲连接，则会启动另外一个后台 goroutine 负责新建连接，并通过 dialc 通道将新建的连接实例送回给 getConn()函数使用。在 Transport.getConn()方法中通过同时监听多个通道来处理多种复杂的场景。例如，在创建的过程中，一个连接被其他 goroutine 释放了等场景。Transport.getConn()方法的具体实现如下：

```
func (t *Transport) getConn(treq *transportRequest, cm connectMethod)
     (*persistConn, error) {
    // 先通过 getIdelConn()方法获取空闲连接，如果有空闲连接，则直接返回
    if pc, idleSince := t.getIdleConn(cm); pc != nil {
        ... ...
        return pc, nil
    }
    ... ...
    dialc := make(chan dialRes) // 该通道用来接收新建连接
```



```
    go func() { // 启动一个单独的 goroutine 来新建连接，并将新建连接写入 dialc 通道
        pc, err := t.dialConn(ctx, cm)
        dialc <- dialRes{pc, err}
    }()

    idleConnCh := t.getIdleConnCh(cm) // 根据 connectMethod 获取对应的 idleConnCh 通道
    select {
    case v := <-dialc:  // 从 dialc 通道中获取了新建连接
        // 检查 TCP 连接是否建立成功，如果建立成功，则直接返回(略)
        return v.pc, nil
        // 检查 TCP 连接建立失败的原因，例如，超时或是取消，并返回相应的异常(略)
    case pc := <-idleConnCh:// 出现空闲连接，此时可能是其他 goroutine 释放了连接或是新建连接
                            // 后未使用
      handlePendingDial() // 对于前面的新建连接，尝试放入 idleConn 中，如果失败，则将其关闭
        return pc, nil  // 返回连接
    case <-req.Cancel: // 请求取消的情况，此时也会通过 handlePendingDial()方法对新建连接进行处理
        handlePendingDial()
        return nil, errRequestCanceledConn
    // 剩余的处理代码比较简单，这里不再赘述，感兴趣的读者可以参考源码进行学习
}
```

上述过程中，Transport.dialConn()方法负责新建连接，其具体实现较为复杂，笔者对其进行了简化，这里简单分析一下其大致流程：

```
func (t *Transport) dialConn(ctx context.Context, cm connectMethod)
        (*persistConn, error) {
    pconn := &persistConn{...} // 新建 persistConn 实例
    // 对于 HTTPS 相关的处理(略)
    conn, err := t.dial(ctx, "tcp", cm.addr()) // 新建连接，省略相关的异常处理
    pconn.conn = conn
    // 省略不同代理类型的处理
    // 省略 HTTPS 相关的处理

    // 这里会将新建的连接的输入流和输出流进行一层封装，得到带缓冲的 Reader 实例和 Writer 实例，
    // 并记录到 persistConn.br 和 bw 字段中
    pconn.br = bufio.NewReader(pconn)
    pconn.bw = bufio.NewWriter(persistConnWriter{pconn})
```



```
    go pconn.readLoop()  // 启动一个单独的 goroutine 专门来读取响应，后面详细介绍
    go pconn.writeLoop()// 启动一个单独的 goroutine 专门来写入请求，后面详细介绍
    return pconn, nil
}
```

在获取到连接之后，Transport.RoundTrip()方法会调用 persistConn.roundTrip()方法将请求发送出去。在开始分析 persistConn.roundTrip()方法之前，先来分析 persistConn 结构体中的关键字段，如下所示。

- t（*Transport 类型）：当前 persistConn 实例关联的 Transport 实例。
- conn（net.Conn 类型）：底层封装的连接。
- br（*bufio.Reader 类型）：它对底层连接的输入流的封装。
- bw（*bufio.Writer 类型）：它对底层连接的输出流的封装。
- reqch（chan requestAndChan 类型）：主 goroutine 向 reqch 通道写入待读取的响应信息，readLoop 从该通道里面接收这些信息并在读取响应之后将其返回。
- writech（chan writeRequest 类型）：主 goroutine 会向 writech 通道中写入待发送的请求，writeLoop 会从该通道中读取待发送的请求。

主 goroutine、readLoop 和 writeLoop 之间的关系如图 4-3 所示。

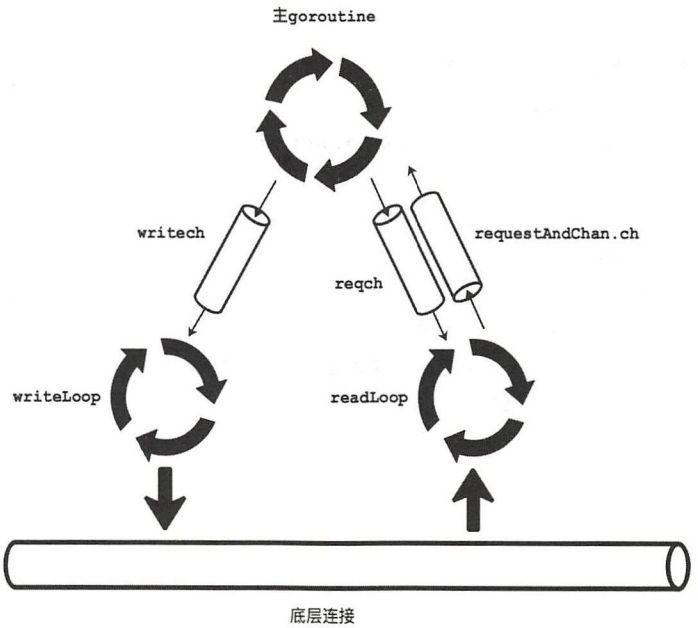


图 4-3



下面来看一下 persistConn.roundTrip()方法，其大致步骤如下：

（1）检测请求是否已经被取消，如果已被取消，则把 persistConn 放入 putOrCloseIdleConn()中处理。

（2）对请求头进行处理。

（3）向 writech 通道写入 writeRequest 实例（其中封装了待发送的请求），上面介绍过 writech 通道的接收者是 writeLoop goroutine，writeLoop goroutine 接收到请求之后就会将请求发送出去，如果发送过程中出现异常，则通过 writeErrCh 通道返回给主 goroutine。

（4）向 reqch 通道写入 requestAndChan 实例（其中封装了请求信息及 readLoop goroutine 返回响应的通道），readloop goroutine 负责读取 reqch 通道中的数据。

（5）persistConn.roundTrip()方法会监听一些写入错误、响应超时、连接关闭等异常情况，并且等待 readLoop goroutine 返回的响应。

下面来看一下 persistConn.roundTrip()方法的大致实现，其中省略了很多复杂场景的处理：

```
func (pc *persistConn) roundTrip(req *transportRequest) (resp *Response, err error) {
  // 步骤 1：检测请求是否已经被取消
  if !pc.t.replaceReqCanceler(req.Request, pc.cancelRequest) {
    pc.t.putOrCloseIdleConn(pc) // 尝试将长连接放入 idleConn 等待复用，若尝试失败，则关闭连接
    return nil, errRequestCanceled
  }
  // 步骤 2：对请求头等部分的一些处理(略)
  writeErrCh := make(chan error, 1)
  // 步骤 3：创建 writeRequest 实例，并写入到 writech 通道中
  pc.writech <- writeRequest{req, writeErrCh, continueCh}

  resc := make(chan responseAndError) // 该通道用来接收 readLoop 中读取到的响应信息
  pc.reqch <- requestAndChan{ // 步骤 4：创建 requestAndChan 实例并写到 reqch 通道中
    req:       req.Request, // 封装了等待响应的请求
    ch:        resc,  // readLoop 通过该通道返回相应的响应
    ... ...  // 暂时省略其他字段
  }

  for {
    select {
    case err := <-writeErrCh:
      // 发送请求出现异常时的相关处理(略)
    case <-respHeaderTimer:
```



```
            // 响应出现超时时进行的异常处理(略)
        case re := <-resc:
            // 收到正常响应之后，先检测有没有异常，如果没有，则返回响应
            return re.res, nil
        // 其他场景的处理(略)
        }
    }
}
```

通过对persistConn.roundTrip()方法的分析可知，响应读取是在readLoop goroutine中完成的。readLoop()方法首先检测当前是否发生了 I/O 错误，然后通过 persistConn.reqch 通道接收主 goroutine 写入的 requestAndChan 实例，并从连接中读取相应的响应数据。正如前面介绍的，requestAndChan 中还封装了 readLoop goroutine 和主 goroutine 进行通信的通道(ch 字段)。之后，readLoop goroutine 会对读取到的响应数据进行检查，并进行反序列化，最后通过 requestAndChan.ch 通道将响应返回给主 goroutine。readLoop()方法处理了多种异常场景，实现比较复杂，这里对其进行简化，只介绍关键步骤，大致实现如下：

```
func (pc *persistConn) readLoop() {
    // 省略一些函数和变量的定义，以及一些 defer 代码
    alive := true
    for alive {
        pc.readLimit = pc.maxHeaderResponseSize()
        _, err := pc.br.Peek(1) // 检测当前是否发生 I/O 异常
        ... ...
        rc := <-pc.reqch // 从 reqch 通道中读取 requestAndChan 实例

        var resp *Response
        if err == nil {
            resp, err = pc.readResponse(rc, trace) // 根据 requestAndChan 读取相应的响应
        }
        if resp.Close || rc.req.Close || resp.StatusCode <= 199 {
            alive = false // 根据请求和响应的信息，决定 readLoop 是否继续执行
        }
        // 对响应进行一系列处理(略)
        select {
        // 将响应封装成 responseAndError 实例，并将其写入 requestAndChan.ch 通道
        case rc.ch <- responseAndError{res: resp}:
```



```
            case <-rc.callerGone:
                return
            }

            // 处理请求取消等场景，其中会释放 persistConn 到连接池中(略)
        }
    }
```

writeLoop()方法的实现相对来说就比较简单了，它会从 writech 通道中读取主 goroutine 写入的待发送请求，然后将其写入底层连接。如果写入出现错误，则通知主 goroutine 进行处理。writeLoop()方法的具体实现如下：

```
func (pc *persistConn) writeLoop() {
    for {
        select {
        case wr := <-pc.writech: // 从 writech 通道中读取待发送的请求
            startBytesWritten := pc.nwrite
            // 将请求写入 persistConn.bw 中，然后调用其 Flush()方法，将请求发送出去，
            // 省略异常处理的相关代码
            err := wr.req.Request.write(pc.bw, pc.isProxy,
                wr.req.extra, pc.waitForContinue(wr.continueCh))
            err = pc.bw.Flush()
            // 对于发送过程中出现的异常，则通过 writeErrCh 通道返回给主 goroutine
            pc.writeErrCh <- err
        case <-pc.closech: // persistConn 关闭，则直接返回
            return
        }
    }
}
```

4.2 etcd-rafthttp模块详解

通过上一章对 etcd-raft 模块的介绍我们知道，etcd-raft 模块并未提供网络层的相关实现，而是将待发送的消息封装进 Ready 实例返回给上层模块，然后由上层模块决定如何将这些消息发送到集群中的其他节点。etcd 将网络层相关的实现独立成一个单独的模块，也就是本节要详细介绍的 etcd-rafthttp 模块。之所以独立出该模块，是为了降低 etcd-raft 模块与网络层实现之间的耦合，降低 etcd-raft 模块和 etcd-rafthttp 模块实现的成本，提高整个程序的可扩展性。



在 etcd 集群中，每个节点启动时都会与集群中的其他节点建立网络连接，这里以三个节点的集群为例，最终形成如图 4-4 所示的网络结构。

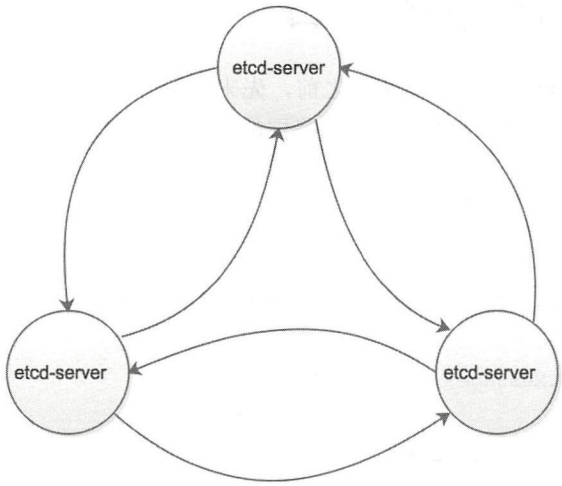


图 4-4

在 etcd 中有多种消息类型，不同类型的消息所能携带的数据大小也不尽相同，其中快照相关消息的数据量就比较大，小至几 KB，大至几 GB 都是有可能的，而 Leader 节点到 Follower 节点之间的心跳消息一般只有几十到几百个字节。因此，etcd 的网络层会创建两个消息传输通道（与 Go 语言中的 Channel 并不是一个层面的东西，请读者注意区分），并对 etcd-raft 模块产生的消息进行分类处理。

上面说的两个消息传输通道也就是本节紧接着要介绍的 Stream 消息通道和 Pipeline 消息通道。这两种消息通道的主要区别在于：Stream 消息通道维护的 HTTP 长连接，主要负责传输数据量较小、发送比较频繁的消息，例如，前面介绍的 MsgApp 消息、MsgHeartbeat 消息、MsgVote 消息等；而 Pipeline 消息通道在传输数据完成后会立即关闭连接，主要负责传输数据量较大、发送频率较低的消息，例如，MsgSnap 消息等。

Stream 消息通道是节点启动后，主动与集群中的其他节点建立的。每个 Stream 消息通道有 2 个关联的后台 goroutine，其中一个用于建立关联的 HTTP 连接，并从连接上读取数据，然后将这些读取到的数据反序列化成 Message 实例，传递到 etcd-raft 模块中进行处理。另外一个后台 goroutine 会读取 etcd-raft 模块返回的消息数据并将其序列化，最后写入 Stream 消息通道。

读者可以回忆一下前面对 raftNode.serveRaft()方法的介绍，在启动 http.Server 时会通过 rafthttp.Transporter.Handler()方法为指定的 URL 路径添加相应的 Handler 实例，其中“/raft/stream/”路径对应的 Handler 为 streamHandler 类型，它负责处理 Stream 消息通道上的请求。相同的，“/raft”路径对应的 Handler 为 pipelineHandler 类型，它负责处理 Pipeline 通道上的请求，“/raft/snapshot”



路径对应的 Handler 为 snapshotHandler 类型，它负责处理 Pipeline 通道上的请求。

4.2.1 rafthttp.Transporter接口

在开始分析这些 Handler 的具体实现之前，先来介绍一下 rafthttp.Transporter 接口，它是 rafthttp 包的核心接口之一，它定义了 etcd 网络层的核心功能，其具体定义如下：

```
type Transporter interface {
    Start() error // 初始化操作

    Handler() http.Handler // 创建 Handler 实例，并关联到指定的 URL 上

    Send(m []raftpb.Message) // 发送消息

    SendSnapshot(m snap.Message) // 发送快照数据

    // 在集群中添加一个节点时，其他节点会通过该方法添加该新加入节点的信息
    AddRemote(id types.ID, urls []string)

    // Peer 接口是当前节点对集群中其他节点的抽象表示，而结构体 peer 则是 Peer 接口的一个具体实现
    // 下面几个方法就是对 Peer 的操作，通过名称即可了解其含义
    AddPeer(id types.ID, urls []string)
    RemovePeer(id types.ID)
    RemoveAllPeers()
    UpdatePeer(id types.ID, urls []string)
    ActiveSince(id types.ID) time.Time

    Stop()// 关闭操作，该方法会关闭全部的网络连接
}
```

另外，还有另一个需要重点介绍的接口——Raft，在前面介绍 raftexample 示例时，也简单提到过该接口，该接口的定义如下：

```
type Raft interface {
    // 将指定的消息实例传递到底层的 etcd-raft 模块进行处理
    Process(ctx context.Context, m raftpb.Message) error

    // 检测指定节点是否从当前集群中移出
```



```
    IsIDRemoved(id uint64) bool

    // 通知底层 etcd-raft 模块，当前节点与指定的节点无法连通
    ReportUnreachable(id uint64)

    // 通知底层的 etcd-raft 模块，快照数据是否发送成功
    ReportSnapshot(id uint64, status raft.SnapshotStatus)
}
```

在后面分析 etcd 服务端的实现时，还会详细介绍 Raft 接口的相关实现。

rafthttp.Transport 是 rafthttp.Transporter 接口具体实现，请读者注意区分这里的 rafthttp.Transport 和前面介绍的 net.http.Transport 接口，两者是完全不同的两个东西。下面来看一下 rafthttp.Transport 中定义的关键字段：

- ID（types.ID 类型）：当前节点自己的 ID。
- URLs（types.URLs 类型）：当前节点与集群中其他节点交互时使用的 URL 地址。
- ClusterID（types.ID 类型）：当前节点所在的集群的 ID。
- Raft（Raft 类型）：Raft 是一个接口，其实现的底层封装了前面介绍的 etcd-raft 模块，当 rafthttp.Transport 收到消息之后，会将其交给 Raft 实例进行处理。
- Snapshotter（*snap.Snapshotter 类型）：Snapshotter 负责管理快照文件，后面会详细介绍其实现。
- streamRt（http.RoundTripper 类型）：Stream 消息通道中使用的 http.RoundTripper 实例。
- pipelineRt（http.RoundTripper 类型）：Pipeline 消息通道中使用的 http.RoundTripper 实例，http.Transport（http.RoundTripper 接口的实现）的工作原理在前面已经介绍过了，这里不再重复介绍。
- peers(map[types.ID]Peer 类型）：Peer 接口是当前节点对集群中其他节点的抽象表示。对于当前节点来说，集群中其他节点在本地都会有一个 Peer 实例与之对应，peers 字段维护了节点 ID 到对应 Peer 实例之间的映射关系。
- remotes（map[types.ID]*remote 类型）：remote 中只封装了 pipeline 实例，remote 主要负责发送快照数据，帮助新加入的节点快速追赶上其他节点的数据。
- prober（probing.Prober 类型）：用于探测 Pipeline 消息通道是否可用。

了解了 rafthttp.Transporter 接口的定义和 rafthttp.Transport 中的各个字段的含义之后，我们再来了解一下 Transport.Start()方法中执行的初始化操作，其具体实现如下：



```
func (t *Transport) Start() error {
    var err error
    // 创建Stream消息通道使用的http.RoundTripper实例,底层实际上是创建http.Transport实例,
    // 这里需要读者注意一下几个参数的设置，分别是：创建连接的超时时间(根据配置指定)、
    // 读写请求的超时时间(默认5s)和keepAlive时间(默认为30s)
    t.streamRt, err = newStreamRoundTripper(t.TLSInfo, t.DialTimeout)
    // 异常处理(略)

    // 创建Pipeline消息通道使用的http.RoundTripper实例
    // 与streamRt不同的是，读写请求的超时时间设置成了永不过期
    t.pipelineRt, err = NewRoundTripper(t.TLSInfo, t.DialTimeout)

    t.remotes = make(map[types.ID]*remote) // 初始化remotes和peers两个map字段
    t.peers = make(map[types.ID]Peer)
    // 该prober实例主要用于探测Pipeline消息通道是否可用
    t.prober = probing.NewProber(t.pipelineRt)
    return nil
}
```

Transport.Handler()方法主要负责创建 Steam 消息通道和 Pipeline 消息通道用到的 Handler 实例，并注册到相应的请求路径上，其具体实现在前面介绍过了，这里不再重复分析。

下面再来看一下 Transport.AddPeer()方法，其主要工作就是创建并启动对应节点的 Peer 实例，具体实现如下：

```
func (t *Transport) AddPeer(id types.ID, us []string) {
    // 检测当前raftthttp.Transport的运行状态(略)
    // 是否已经与指定ID的节点建立了连接(略)
    urls, err := types.NewURLs(us)   // 解析us切片中指定URL连接，异常处理(略)
    // 创建指定节点对应的Peer实例，其中会相关的Stream消息通道和Pipeline消息通道
    t.peers[id] = startPeer(t, urls, id, fs)
    // 每隔一段时间，prober会向该节点发送探测消息，检测对端的健康状况
    addPeerToProber(t.prober, id.String(), us)
}
```

在 rafthttp.Transport 中实现的 RemovePeer()方法会调用 peer.stop()方法关闭底层的连接，同时会停止定时发送的探测消息，RemoveAllPeers()方法的实现同理；UpdatePeer()方法则会调用 peer.update()方法更新对端暴露的 URL 地址，同时更新探测消息发送的目标地址。另外，Transport



还实现了 rafthttp.peerGetter 接口，该接口中定义了一个 Get()方法用于获取指定节点对应的 Peer 实例。上述四个方法的具体实现比较简单，代码就不在贴出来了，感兴趣的读者请参考源码进行学习。

Transport.Send()方法负责发送指定的 raftpb.Message 消息，其中首先尝试使用目标节点对应的 Peer 实例发送消息，如果没有找到对应的 Peer 实例，则尝试使用对应的 remote 实例发送消息。Transport.Send()方法的具体实现如下：

```
func (t *Transport) Send(msgs []raftpb.Message) {
    for _, m := range msgs { // 遍历 msgs 切片中的全部消息
        // 根据 raftpb.Message.To 字段，获取目标节点对应的 Peer 实例
        to := types.ID(m.To)
        t.mu.RLock() // 加锁
        p, pok := t.peers[to]
        g, rok := t.remotes[to]
        t.mu.RUnlock()// 解锁

        if pok { // 如果存在对应的 Peer 实例，则使用 Peer 发送消息
            if m.Type == raftpb.MsgApp { // 统计信息
                t.ServerStats.SendAppendReq(m.Size())
            }
            // 记录监控信息(略)
            p.send(m) // 通过 peer.send()方法完成消息的发送
            continue
        }

        if rok { // 如果指定节点 ID 不存在对应的 Peer 实例，则尝试使用查找对应 remote 实例
            g.send(m) // 通过 remote.send()方法完成消息的发送
            continue
        }
        // 执行到这里表示无法找到 raftpb.Message 的目标节点，则记录日志信息(略)
    }
}
```

Transport.SendSnapshot()方法负责发送指定的 snap.Message 消息（其中封装了对应的 MsgSnap 消息实例及其他相关信息）。该方法是通过调用 peer.sendSnap()方法完成的，实现比较简单，在下一小节介绍 peer.sendSnap()方法时再详细介绍其具体实现。



4.2.2 Peer接口

通过上一小节的分析我们知道，rafthttp.Transport 中的多个方法都是通过调用 Peer 实例的相应方法实现的。本小节先来介绍一下 Peer 接口，其具体定义如下：

```
type Peer interface {
    // 发送单个消息，该方法是非阻塞的，如果出现发送失败，则会将失败信息报告给底层的 Raft 接口
    send(m raftpb.Message)

    sendSnap(m snap.Message) // 发送 snap.Message，其他行为与上面的 send() 方法类似

    update(urls types.URLs) // 更新对应节点暴露的 URL 地址

    // 将指定的连接与当前 Peer 绑定，Peer 会将该连接作为 Stream 消息通道使用
    // 当 Peer 不再使用该连接时，会将该连接关闭
    attachOutgoingConn(conn *outgoingConn)

    stop()   // 关闭当前 Peer 实例，会关闭底层的网络连接
}
```

结构体 rafthttp.peer 是 Peer 接口的具体实现，其中各字段的含义如下。

- id（types.ID 类型）：该 peer 实例对应的节点的 ID。
- r（Raft 类型）：Raft 接口，在 Raft 接口实现的底层封装了 etcd-raft 模块。
- picker（*urlPicker 类型）：每个节点可能提供了多个 URL 供其他节点访问，当其中一个访问失败时，我们应该可以尝试访问另一个。而 urlPicker 提供的主要功能就是在这些 URL 之间进行切换。
- writer（*streamWriter 类型）：负责向 Stream 消息通道写入消息。
- msgAppReader（*streamReader 类型）：负责从 Stream 消息通道读取消息。
- pipeline（*pipeline 类型）：Pipeline 消息通道。
- snapSender（*snapshotSender 类型）：负责发送快照数据。
- recvc（chan raftpb.Message 类型）：从 Stream 消息通道中读取到消息之后，会通过该通道将消息交给 Raft 接口，然后由它返回给底层 etcd-raft 模块进行处理。
- propc（chan raftpb.Message 类型）：从 Stream 消息通道中读取到 MsgProp 类型的消息之后，会通过该通道将 MsgProp 消息交给 Raft 接口，然后由它返回给底层 etcd-raft



模块进行处理。

- paused（bool 类型）：是否暂停向对应节点发送消息。

上述字段之间协同工作的原理如图 4-5 所示。

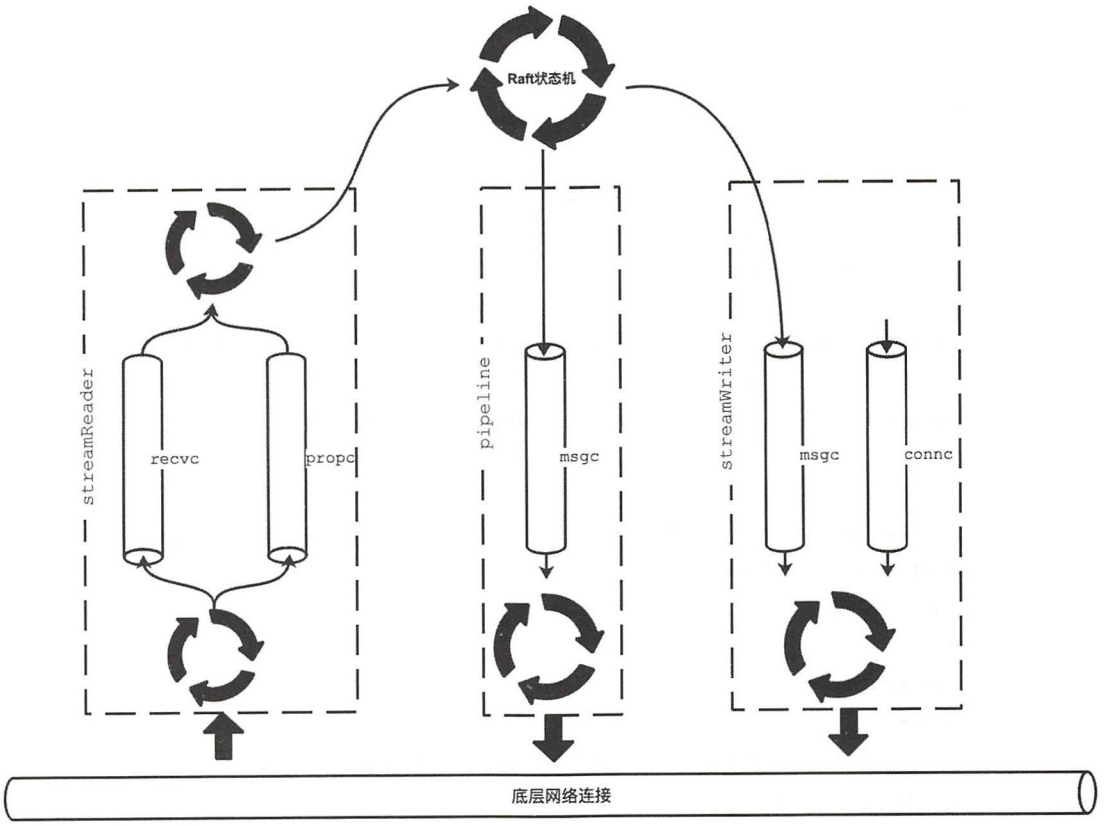


图 4-5

在 rafthttp.peer.startPeer()方法中完成了初始化上述字段的工作，同时也启动了关联的后台 goroutine。该方法的具体实现如下，其中省略了一些异常处理的代码：

```
func startPeer(transport *Transport, urls types.URLs, peerID types.ID,
    fs *stats.FollowerStats) *peer {
  picker := newURLPicker(urls)    // 根据节点提供的 URL 创建 urlPicker
  r := transport.Raft             // 底层的 Raft 状态机
  pipeline := &pipeline{ // 创建 pipeine 实例
    peerID: peerID,
    tr: transport,
```



```
    picker: picker,
    status: status,
    raft: r,
}
pipeline.start() // 启动 pipeline

p := &peer{// 创建 Peer 实例
    id: peerID,
    r: r,
    picker: picker,
    writer: startStreamWriter(peerID, status, fs, r), // 创建并启动 streamWriter
    pipeline: pipeline,
    snapSender: newSnapshotSender(transport, picker, peerID, status),
    recvc: make(chan raftpb.Message, recvBufSize), // 创建 recvc 通道，注意缓冲区大小
    // 创建 propc 通道，注意缓冲区大小
    propc: make(chan raftpb.Message, maxPendingProposals),
}

// 启动单独的 goroutine，它主要负责将 recvc 通道中读取消息，该通道中的消息就是从对端节点发送过
// 来的消息，然后将读取到的消息交给底层的 Raft 状态机进行处理
go func() {
    for {
        select {
        case mm := <-p.recvc: // 从 recvc 通道中获取连接上读取到的 Message
            // 将 Message 交给底层 Raft 状态机处理，异常处理(略)
            if err := r.Process(ctx, mm); ...
        }
    }
}()

// 在底层 Raft 状态机处理 MsgProp 类型的 Message 时，可能会阻塞，所以启动单独的 goroutine 来处理
go func() {
    for {
        select {
        case mm := <-p.propc: // 从 propc 通道中获取 MsgProp 类型的 Message
            // 将 Message 交给底层 Raft 状态机处理，异常处理(略)
            if err := r.Process(ctx, mm); ...
        }
```



```
        }
    }()

    // 创建并启动 streamReader 实例，它主要负责从 Stream 消息通道上读取消息
    p.msgAppReader = &streamReader{
        peerID: peerID,
        typ:    streamTypeMessage,
        tr:     transport,
        picker: picker,
        recvc:  p.recvc,
        propc:  p.propc,
    }
    p.msgAppReader.start()
    return p
}
```

了解了 Peer 实例的初始化操作之后，再来看其核心方法。首先是 peer.send()方法，前面分析的 Transport.Send()方法就是通过调用该方法实现消息发送功能的，具体实现如下：

```
func (p *peer) send(m raftpb.Message) {
    // 检测 paused 字段，是否暂停对指定节点发送消息(略)
    // 根据消息的类型选择合适的消息通道，peer.pick()方法下面会详细介绍
    writec, name := p.pick(m)
    select {
    case writec <- m: // 将 Message 写入 writec 通道中，等到发送
    default:
        // 如果发送出现阻塞，则将信息报告给底层 Raft 状态机，这里也会根据消息类型选择合适的报告方法
        p.r.ReportUnreachable(m.To)
        if isMsgSnap(m) {
            p.r.ReportSnapshot(m.To, raft.SnapshotFailure)
        }
        // 记录日志(略)
    }
}
```

在 peer.pick()方法中，会根据消息的类型选择合适的消息通道，并返回相应的通道供 send()方法写入待发送的消息，具体实现如下：



```
func (p *peer) pick(m raftpb.Message) (writec chan<- raftpb.Message, picked string) {
    var ok bool
    // 如果是 MsgSnap 类型的消息，则返回 Pipeline 消息通道对应的 Channel，否则返回 Stream 消息通
    // 道对应的 Channel，如果 Stream 消息通道不可用，则使用 Pipeline 消息通道发送所有类型的消息
    if isMsgSnap(m) {
        return p.pipeline.msgc, pipelineMsg
    } else if writec, ok = p.writer.writec(); ok {
        return writec, streamMsg
    }
    return p.pipeline.msgc, pipelineMsg
}
```

Peer 接口及其实现 peer 结构体的内容暂时先介绍到这里，Peer 的其他方法在后面使用到的时候再进行分析。

4.2.3 pipeline

正如本章前面所介绍的，pipeline 负责传输快照数据。我们首先来分析 pipeline 的核心字段。

- peerID（types.ID 类型）：该 pipeline 对应节点的 ID。
- tr（*Transport 类型）：关联的 rafthttp.Transport 实例。
- raft（Raft 类型）：底层的 Raft 实例。
- msgc（chan Message 类型）：pipeline 实例从该通道中获取待发送的消息。
- wg（sync.WaitGroup 类型）：负责同步多个 goroutine 结束。每个 pipeline 实例会启动多个后台 goroutine（默认值是 4 个）来处理 msgc 通道中的消息，在 pipeline.stop()方法中必须等待这些 goroutine 都结束（通过 wg.Wait()方法实现），才能真正关闭该 pipeline 实例。

pipeline 的大致工作原理如图 4-6 所示。



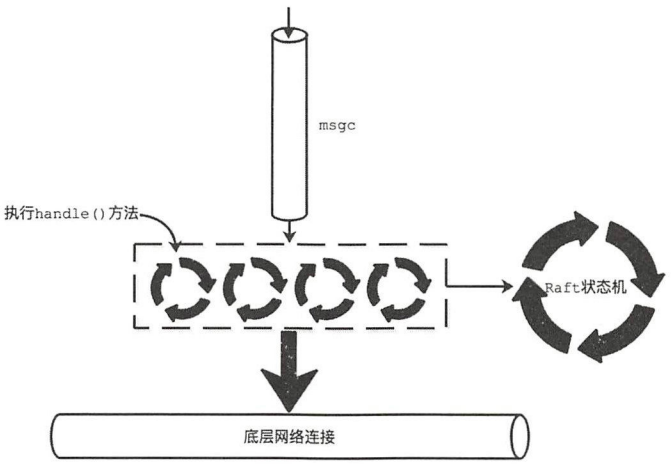


图 4-6

在 pipeline.start()方法中会初始化上述字段，同时会启动用来发送消息的后台 goroutine，其具体实现如下：

```
func (p *pipeline) start() {
  // 注意缓存，默认是 64，主要是为了防止瞬间网络延迟造成消息丢失
  p.msgc = make(chan raftpb.Message, pipelineBufSize)
  p.wg.Add(connPerPipeline) // 初始化 sync.WaitGroup
  for i := 0; i < connPerPipeline; i++ { // 启动用于发送消息的 goroutine（默认是 4 个）
    go p.handle()
  }
}
```

在 pipeline.handle()方法中会从 msgc 通道中读取待发送的 Message 消息，然后调用 pipeline.post()方法将其发送出去，发送结束之后会调用底层 Raft 接口的相应方法报告发送结果。pipeline.handle()方法的具体实现如下：

```
func (p *pipeline) handle() {
  defer p.wg.Done() // handle()方法执行完成，也就是当前这个 goroutine 结束
  for {
    select {
    case m := <-p.msgc: // 获取待发送的 MsgSnap 类型的 Message
      start := time.Now()
      err := p.post(pbutil.MustMarshal(&m)) // 将消息序列化，然后创建 HTTP 请求并发送出去
      if err != nil {
```



```
            // 其他的异常处理代码(略)
            // 向底层的 Raft 状态机报告失败信息，Raft 接口的内容后面详细介绍
            p.raft.ReportUnreachable(m.To)
            if isMsgSnap(m) {
              p.raft.ReportSnapshot(m.To, raft.SnapshotFailure)
            }
            continue
          }
          // 向底层的 Raft 状态机报告发送成功的信息
          if isMsgSnap(m) {
            p.raft.ReportSnapshot(m.To, raft.SnapshotFinish)
          }
        case <-p.stopc:
          return
        }
      }
    }
```

pipeline.post()方法是真正完成消息发送的地方，其中会启动一个后台 goroutine 监听控制发送过程及获取发送结果，具体实现如下：

```
func (p *pipeline) post(data []byte) (err error) {
  u := p.picker.pick() // 获取对端暴露的 URL 地址
  // 创建 HTTP POST 请求
  req := createPostRequest(u, RaftPrefix,
    bytes.NewBuffer(data), "application/protobuf",
      p.tr.URLs, p.tr.ID, p.tr.ClusterID)

  done := make(chan struct{}, 1) // 主要用于通知下面的 goroutine 请求是否已经发送完成
  ctx, cancel := context.WithCancel(context.Background())
  req = req.WithContext(ctx)
  go func() { // 该 goroutine 主要用于监听请求是否需要取消
    select {
    case <-done:
    case <-p.stopc: // 如果在请求的发送过程中，pipeline 被关闭，则取消该请求
      waitSchedule()
      cancel()// 取消请求
    }
```



```
    }()
    // 发送上述 HTTP POST 请求，并获取到对应的响应，省略异常处理的相关代码
    resp, err := p.tr.pipelineRt.RoundTrip(req)
    done <- struct{}{}   // 通知上述 goroutine，请求已经发送完毕

    b, err := ioutil.ReadAll(resp.Body)   // 读取 HttpResponse.Body 内容
    if err != nil {
        p.picker.unreachable(u) // 出现异常时，则将该 URL 标识为不可用，再尝试其他 URL 地址
        return err
    }
    resp.Body.Close()
    err = checkPostResponse(resp, b, req, p.peerID) // 检测响应的内容，异常处理(略)
    return nil
}
```

通过上面的分析我们可以大致了解 pipeline 发送消息的大致原理。在前面介绍 rafthttp.Transport.Handler()方法时提到了 pipelineHandler 这个 Handler 实现，它负责接收 pipeline 发送的数据，在后面遇到 pipelineHandler 时，会详细分析其具体实现。

4.2.4 streamWriter实例

在 peer.start()方法中，除了创建并启动 pipeline 实例，还会创建并启动了一个 sreamWriter 实例。从命名上可以猜出其主要功能就是向 Stream 消息通道写入消息，本节就来详细介绍 streamWriter 的具体实现。先来看一下 streamWriter 的核心字段，如下所示。

- peerID（types.ID 类型）：对端节点的 ID。
- r（Raft 类型）：底层的 Raft 实例。
- closer（io.Closer 类型）：负责关闭底层的长连接。
- working（bool 类型）：负责标识当前的 streamWriter 是否可用（底层是否关联了相应的网络连接）。
- msgc（chan Message 类型）：通过前面对 Peer 的分析可知，Peer 会将待发送的消息写入该通道，streamWriter 则从该通道中读取消息并发送出去。
- connc（chan *outgoingConn 类型）：通过该通道获取当前 streamWriter 实例关联的底层网络连接，outgoingConn 其实是对网络连接的一层封装，其中记录了当前连接使用的协议版本，以及用于关闭连接的 Flusher 和 Closer 等信息。



StreamWriter 的大致工作原理如图 4-7 所示。

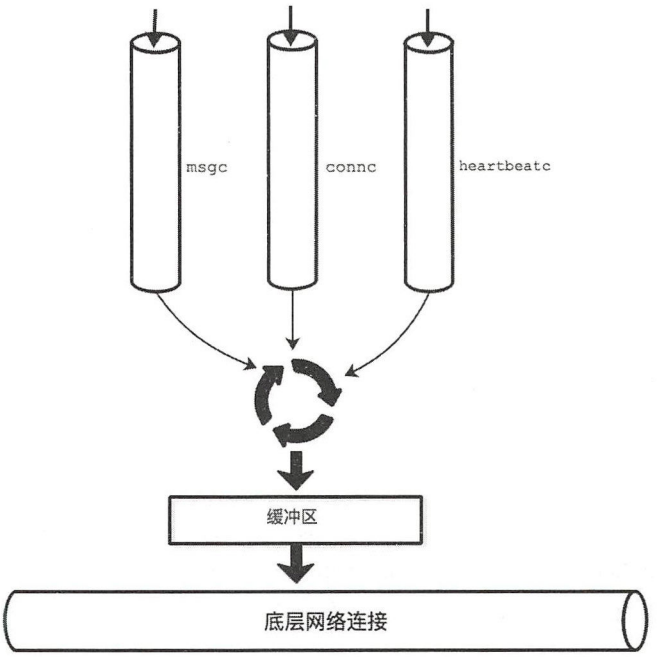


图 4-7

在 peer.start()方法中是通过调用 startStreamWriter()方法初始化上述字段并启动 streamWriter 实例的，其中还启动了一个后台 goroutine 来执行 streamWriter.run()方法。在 streamWriter.run()方法中，主要完成了下面三件事情：

（1）当其他节点主动与当前节点创建连接（即 Stream 消息通道底层使用的网络连接）时，该连接实例会写入对应 peer.writer.connc 通道，在 streamWriter.run()方法中会通过该通道获取该连接实例并进行绑定，之后才能开始后续的消息发送。

（2）定时发送心跳消息，该心跳消息并不是前面介绍 etcd-raft 模块时提到的 MsgHeartbeat 消息，而是为了防止底层连接超时的消息，后面会详细介绍该消息的处理过程。

（3）发送除心跳消息外的其他类型的消息。

streamWriter.run()方法虽然有点长，但是这样拆开来看就不是很复杂了。streamWriter.run()方法的具体实现如下：

```
func (cw *streamWriter) run() {
  var (
    msgc chan raftpb.Message  // 指向当前 streamWriter.msgc 字段
    // 定时器会定时向该通道发送信号，触发心跳消息的发送，该心跳消息与后面介绍的 Raft 的心跳消息
```



```
    // 有所不同，该心跳消息的主要目的是为了防止连接长时间不用断开的
    heartbeatc <-chan time.Time
    t          streamType    // 用来记录消息的版本信息
    enc        encoder       // 编码器，负责将消息序列化并写入连接的缓冲区
    flusher    http.Flusher // 负责刷新底层连接，将数据真正发送出去
    batched    int           // 当前未 Flush 的消息个数
)
tickc := time.NewTicker(ConnReadTimeout / 3) // 发送心跳消息的定时器
unflushed := 0 // 未 Flush 的字节数

for {
    select {
    // 当其他节点主动与当前节点创建 Stream 消息通道时，会先通过 StreamHandler 的处理，
    // StreamHandler 会通过 attach()方法将连接写入对应 peer.writer.connc 通道，
    // 而当前的 goroutine 会通过该通道获取连接，然后开始发送消息
    case conn := <-cw.connc: // 获取与当前 streamWriter 实例绑定的底层连接
        cw.mu.Lock()
        t = conn.t // 获取该连接底层发送的消息版本，并创建相应的 encoder 实例
        switch conn.t {
        case streamTypeMessage: // 这里只关注最新版本，忽略其他版本
            // 将 http.ResponseWriter 封装成 messageEncoder，上层调用通过 messageEncoder 实
            // 例完成消息发送
            enc = &messageEncoder{w: conn.Writer}
        }
        flusher = conn.Flusher // 记录底层连接对应的 Flusher
        unflushed = 0 // 重置未 Flush 的字节数
        cw.status.activate()    // peerStatus.active 设置为 true
        cw.closer = conn.Closer // 记录底层连接对应的 Flusher
        cw.working = true       // 标识当前 streamWriter 正在运行

        // 更新 heartbeatc 和 msgc 两个通道，自此之后，才能发送消息
        heartbeatc, msgc = tickc.C, cw.msgc

    case <-heartbeatc: // 定时器到期，触发心跳消息
        // 通过 encoder 将心跳消息
        err := enc.encode(&linkHeartbeatMessage)
        unflushed += linkHeartbeatMessage.Size() // 增加未 Flush 出去的字节数
        // 若没有异常，则使用 flusher 将缓存的消息全部发送出去，并重置 batched 和 unflushed 两个
```



```
            // 统计变量(略)
            cw.close()// 若发生异常，则关闭 streamWriter，会导致底层连接的关闭
            // 将 heartbeatc 和 msgc 两个通道清空，后续就不会再发送心跳消息和其他类型的消息了
            heartbeatc, msgc = nil, nil

        case m := <-msgc: // peer 向 streamWriter.msgc 写入待发送的消息
            err := enc.encode(&m) // 将消息序列化并写入底层连接
            if err == nil { // 若没有异常，则递增 unflushed 变量
                unflushed += m.Size()
                // msgc 通道中的消息全部发送完成或是未 Flush 的消息较多，则触发 Flush，
                // 否则只是递增 batched 变量
                if len(msgc) == 0 || batched > streamBufSize/2 {
                    flusher.Flush()
                    sentBytes.WithLabelValues(cw.peerID.String()).Add(float64(unflushed))
                    unflushed = 0
                    batched = 0
                } else {
                    batched++
                }
                continue
            }
            // 若发生异常，则关闭 streamWriter，清空 heartbeatc 和 msgc 两个通道，这与发送心跳消息
            // 异常时的处理一样(略)
        }
    }
}
```

streamWriter 中另一个需要读者了解的方法是 attach()方法，该方法会接收 outgoingConn 实例并写入 streamWriter.connc 通道中，peer.attachOutgoingConn()方法就是通过调用该方法实现的。在后面介绍的 streamHandler 中，也就是通过调用 peer.attachOutgoingConn()方法将底层网络连接传递到 streamWriter 中的。streamWriter.attach()方法和 peer.attachOutgoingConn()方法的具体实现如下：

```
func (p *peer) attachOutgoingConn(conn *outgoingConn) {
    var ok bool
    switch conn.t {
    case streamTypeMessage:          // 这里只关注最新版本的实现，省略其他版本的代码
        ok = p.writer.attach(conn) // 将 conn 实例交给 streamWriter 处理
```



```
    default:
        plog.Panicf("unhandled stream type %s", conn.t)
    }
    if !ok {  // 出现异常则关闭连接
        conn.Close()
    }
}

// 下面是 streamWriter.attach()方法的具体实现
func (cw *streamWriter) attach(conn *outgoingConn) bool {
    select {
    case cw.connc <- conn: // 将 conn 实例写入 streamWriter.connc 通道中
        return true
    ... ...
    }
}
```

4.2.5　streamReader实例

在 peer.start()方法中，除了创建前面介绍的 pipeline 和 steamWriter，还创建并启动了一个 sreamReader 实例。从该结构体的命名和前面对 streamWriter 的介绍，读者应该可以猜出其主要功能是从 Stream 通道中读取消息。在本小节中，将详细分析 streamReader 的具体实现。下面先来分析 streamReader 的核心字段。

- peerID（types.ID 类型）：对应节点的 ID。
- typ（streamType 类型）：关联的底层连接使用的协议版本信息。
- tr（*Transport 类型）：关联的 rafthttp.Transport 实例。
- picker（*urlPicker 类型）：用于获取对端节点的可用的 URL。
- recvc（chan<- Message 类型）：在前面介绍的 peer.startPeer()方法中提到，创建 streamReader 实例时是使用 peer.recvc 通道初始化该字段的，其中还会启动一个后台 goroutine 从 peer.recvc 通道中读取消息。在下面分析中会看到，从对端节点发送来的非 MsgProp 类型的消息会首先由 streamReader 写入 recvc 通道中，然后由 peer.start() 启动的后台 goroutine 读取出来，交由底层的 etcd-raft 模块进行处理。
- propc（chan<- raftpb.Message 类型）：该通道与上面介绍的 recvc 通道类似，只不过其中接收的是 MsgProp 类型的消息。
- paused（bool 类型）：是否暂停读取数据。



仅供非商业用途或交流学习使用

streamReader 的大致工作原理如图 4-8 所示。

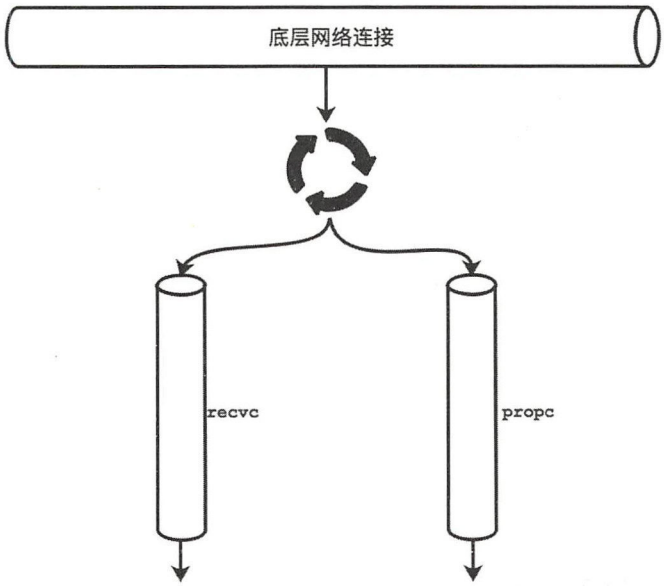


图 4-8

在 streamReader.start()方法中会启动一个单独的后台 goroutine 来执行 streamReader.run()方法，与 streamWriter 类似，它也是 streamReader 的核心。streamReader.run()方法的具体实现如下：

```
func (cr *streamReader) run() {
  t := cr.typ // 获取使用的消息版本
  for {
    // 向对端节点发送一个 GET 请求，然后获取并返回相应的 ReadCloser，异常处理(略)
    // streamReader.dial()方法的具体实现在下面详细介绍
    rc, err := cr.dial(t)
    ... ...
    // 如果未出现异常，则开始读取对端返回的消息，并将读取到的消息写入 streamReader.recvc 通道中
    // streamReader.decodeLoop()方法的具体实现后面详细介绍
    err := cr.decodeLoop(rc, t)
    // streamReader.decodeLoop()方法退出可能是对端主动关闭连接，此时需要等待 100ms
    // 后再创建新连接，这主要是为了防止频繁重试
    select {
    case <-time.After(100 * time.Millisecond):
    case <-cr.stopc: // 如果是正常停止，则关闭 done 通道后返回
      close(cr.done)
```



仅供非商业用途或交流学习使用

```
            return
        }
    }
}
```

streamReader.dial()方法主要负责与对端节点建立连接，其中包含了多种异常情况的处理，这里重点分析其建立连接的主流程，具体实现如下：

```
func (cr *streamReader) dial(t streamType) (io.ReadCloser, error) {
    u := cr.picker.pick() // 获取对端节点暴露的一个URL
    uu := u
    // 根据使用协议的版本和节点ID创建最终的URL地址
    uu.Path = path.Join(t.endpoint(), cr.tr.ID.String())
    // 创建一个GET请求
    req, err := http.NewRequest("GET", uu.String(), nil)
    // 设置HTTP请求头(略)

    setPeerURLsHeader(req, cr.tr.URLs) // 将当前节点暴露的URL也一起发送给对端节点
    // 检测当前streamReader是否被关闭，若已被关闭，则返回异常(略)
    // 通过前面介绍的rafthttp.Transport.streamRt与对端建立连接，异常处理(略)
    resp, err := cr.tr.streamRt.RoundTrip(req) // 发送请求，ping-pong
    // 解析HTTP响应，检测版本信息(略)
    switch resp.StatusCode { // 根据HTTP的响应码进行处理，这里我们只关注正常情况(StatusOK)
    case http.StatusOK:
        return resp.Body, nil
    // 其他异常的情况都省略掉
    }
}
```

decodeLoop()方法是streamReader中的核心方法，它会从底层的网络连接读取数据并进行反序列化，之后将得到的消息实例写入recvc通道（或propc通道）中，等待Peer进行处理，其具体实现如下：

```
func (cr *streamReader) decodeLoop(rc io.ReadCloser, t streamType) error {
    var dec decoder
    cr.mu.Lock()
    // 根据使用的协议版本创建对应的decoder实例，这里只关注协议的最新版本，其他版本的代码省略
    switch t {
```



仅供非商业用途或交流学习使用

```
        case streamTypeMessage:
            dec = &messageDecoder{r: rc} // messageDecoder 主要负责从连接中读取数据
            ... ...
        }
        // 检测 streamReader 是否已经关闭了，若关闭则返回异常(略)
        for {
            // 从底层连接中读取数据，并反序列化成 raftpb.Message 实例，异常处理(略)
            m, err := dec.decode()
            ... ...
            paused := cr.paused
            if paused { // 检测是否已经暂停从该连接上读取消息
                continue
            }

            // 忽略连接层的心跳消息，注意与后面介绍的 Raft 的心跳消息进行区分
            if isLinkHeartbeatMessage(&m) {
                continue
            }

            // 根据读取到的消息类型，选择对应通道进行写入
            recvc := cr.recvc
            if m.Type == raftpb.MsgProp {
                recvc = cr.propc
            }
            select {
            case recvc <- m: // 将消息写入对应的通道中，之后会交给底层的 Raft 状态机进行处理
            ... ... // recvc 通道满了之后，只能丢弃消息，并打印日志(略)
        }
    }
```

到此为止，streamReader 中核心方法的具体实现就介绍完了。

4.2.6 snapshotSender

在 etcd 提供的 v3 版本中，开始使用 snapshotSender 传输快照数据，在前面介绍的 startPeer() 函数中会创建 snapshotSender 实例。snapshotSender 的实现与前面介绍的 pipeline 类似，下面先介绍其核心字段的含义。



仅供非商业用途或交流学习使用

- from 和 to（types.ID 类型）：记录当前节点的 ID 及对端节点的 ID。
- cid（types.ID 类型）：记录当前集群的 ID。
- tr（*Transport 类型）：关联的 Transport 实例。
- picker（*urlPicker 类型）：负责获取对端节点可用的 URL 地址。
- r（Raft 类型）：底层的 Raft 状态机。

前面介绍的 peer 实现了 Peer 接口，其 sendSnap()方法的实现中会单独启动一个线程并调用 snapshotSender.send()方法完成快照数据的发送，具体实现如下：

```
func (p *peer) sendSnap(m snap.Message) {
  go p.snapSender.send(m)
}

func (s *snapshotSender) send(merged snap.Message) {
  m := merged.Message
  body := createSnapBody(merged) // 根据传入的 snap.Message 实例创建请求 body
  defer body.Close()
  u := s.picker.pick() // 选择对端节点可用的 URL 地址
  // 创建请求，注意，这里请求的路径是"/raft/snapshot"，而 pipeline 发出的请求对应的路径是
  // "/raft"
  req := createPostRequest(u, RaftSnapshotPrefix, body,
    "application/octet-stream", s.tr.URLs, s.from, s.cid)

  err := s.post(req) // 发送请求
  defer merged.CloseWithError(err)
  // 异常处理过程与 pipeline 的实现类似，会将此次请求的 URL 设置为不用，然后调用
  // ReportUnreachable()方法通知底层的 etcd-raft 模块对端节点不可达，
  // 再调用 ReportSnapshot()方法将此次发送失败的信息通知底层 etcd-raft 模块，省略具体代码

  // 通知底层的 etcd-raft 模块，此次成功发送快照数据
  s.r.ReportSnapshot(m.To, raft.SnapshotFinish)
}
```

与 pipeline 类似，snapshotSender.post()方法是真正完成快照数据发送的地方，其具体实现如下：

```
func (s *snapshotSender) post(req *http.Request) (err error) {
  // 对请求进行一系列的封装(略)
```



仅供非商业用途或交流学习使用

```
    // 该通道用于返回该请求对应的响应(或是异常信息)
    result := make(chan responseAndError, 1)

    go func() { // 启动一个单独的 goroutine，用于发送请求并读取响应
        resp, err := s.tr.pipelineRt.RoundTrip(req)// 发送请求
        if err != nil {
            // 将异常写入 result 通道中
            result <- responseAndError{resp, nil, err}
            return
        }
        // 超时处理
        time.AfterFunc(snapResponseReadTimeout, func() { httputil.GracefulClose(resp) })
        body, err := ioutil.ReadAll(resp.Body) // 读取响应数据
        result <- responseAndError{resp, body, err} // 将响应数据进行封装，并写入 result 通道中
    }()

    select {
    case r := <-result: // 读取 result 通道，获取响应信息，省略相关的异常处理代码
        return checkPostResponse(r.resp, r.body, req, s.to)
    }
}
```

4.3 Handler实例

在前面介绍的 rafthttp.Transport.Handler()方法中，会创建多个 Handler 实例并与指定的 URL 路径关联，本节中我们重点关注其中的 pipelineHandler 和 streamHandler 两个 Handler 的具体实现。

4.3.1 pipelineHandler

在前面分析的 pipeline.handle()方法和 pipeline.post()方法中，我们只看到了创建连接和发送请求的逻辑，本小节重点介绍在对端节点的 pipelineHandler 中是如何读取快照消息的。下面先来看一下 pipelineHandler 中各个字段的含义。

- tr（Transporter 类型）：当前 pipeline 实例关联的 rafthttp.Transport 实例。
- r（Raft 类型）：底层的 Raft 实例。
- cid（types.ID 类型）：当前集群的 ID。



仅供非商业用途或交流学习使用

正如前面介绍的那样，在 pipelineHandler 中实现了 http.Server.Handler 接口的 ServeHTTP() 方法，也是其处理请求的核心方法。pipelineHandler.ServeHTTP()方法通过读取对端节点发来的请求得到相应的消息实例，然后将其交给底层的 etcd-raft 模块进行处理，该方法的具体大致实现如下：

```
func (h *pipelineHandler) ServeHTTP(w http.ResponseWriter, r *http.Request) {
    // 首先进行一系列的检查，例如：检查请求的 Method 是否为 POST，检测集群 ID 是否合法，等等(略)

    // 限制每次从底层连接读取的字节数上线，默认是 64KB，因为快照数据可能非常大，为了防止读取超时，
    // 只能每次读取一部分数据到缓冲区中，最后将全部数据拼接起来，得到完整的快照数据
    limitedr := pioutil.NewLimitedBufferReader(r.Body, connReadLimitByte)
    b, err := ioutil.ReadAll(limitedr) // 读取 HTTP 请求的 Body 的全部内容，异常处理(略)

    var m raftpb.Message
    // 反序列化得到 raftpb.Message 实例，异常处理(略)
    if err := m.Unmarshal(b); err != nil ...

    // 将读取到的消息实例交给底层的 Raft 状态机进行处理，省略异常处理的相关实现
    if err := h.r.Process(context.TODO(), m); err != nil ...

    w.WriteHeader(http.StatusNoContent) // 向对端节点返回合适的状态码，表示请求已经被处理
}
```

4.3.2　streamHandler

在前面分析 streamWriter 时介绍了向对端消息发送的基本逻辑，在分析 streamReader 时，我们大致了解了建立连接和读取对端消息的过程。本小节主要介绍 streamHandler，它主要负责在接收到对端的网络连接之后，将其与对应的 streamWriter 实例进行关联。这样，streamWriter 就可以开始向对端节点发送消息了。下面来分析 streamHandler 中各个字段的含义。

- tr（*Transport 类型）：关联的 rafthttp.Transport 实例。
- peerGetter（peerGetter 类型）：peerGetter 接口在前面简单介绍过，其接口中的 Get() 方法会根据指定的节点 ID 获取对应的 peer 实例。
- r（Raft 类型）：底层的 Raft 实例。
- id（types.ID 类型）：当前节点的 ID。
- cid（types.ID 类型）：当前集群的 ID。



仅供非商业用途或交流学习使用

与 pipelineHandler 类似，streamHandler 的核心也是 ServeHTTP()方法，其大致实现如下：

```
func (h *streamHandler) ServeHTTP(w http.ResponseWriter, r *http.Request) {
    // 检测请求 Method 是否为 GET，检测集群的 ID(略)
    var t streamType
    switch path.Dir(r.URL.Path) { // 确定使用的协议版本，这里只关注最新版本
    case streamTypeMessage.endpoint():
        t = streamTypeMessage
    default:
        ...
    }
    // 获取对端节点的 ID，异常处理(略)
    fromStr := path.Base(r.URL.Path)
    from, err := types.IDFromString(fromStr)
    p := h.peerGetter.Get(from) // 根据对端节点 ID 获取对应的 Peer 实例

    wto := h.id.String() // 获取当前节点的 ID
    // 检测请求的目标节点是否为当前节点(略)

    w.WriteHeader(http.StatusOK) // 返回 200 状态码
    w.(http.Flusher).Flush()     // 调用 Flush()方法将响应数据发送到对端节点

    conn := &outgoingConn{// 创建 outgoingConn 实例
        t: t, // 协议版本
        Writer: w,
        Flusher: w.(http.Flusher),
        Closer: c,
    }
    // 将 outgoingConn 实例与对应的 streamWriter 实例绑定，peer.attachOutgoingConn()方法
    // 在前面已经介绍过了，这里不再赘述
    p.attachOutgoingConn(conn)
    ... ...
}
```

4.3.3 snapshotHandler

snapshotHandler 与前面介绍的 pipelineHandler 类似，用来接收对端节点发来的快照数据。snapshotHandler 中各个字段的含义如下所示。



仅供非商业用途或交流学习使用

- tr（Transporter 类型）：关联的 Transporter 实例。
- r（Raft 类型）：底层的 Raft 实例。
- snapshotter（*snap.Snapshotter 类型）：负责将快照数据保存到本地文件中，Snapshotter 的具体实现在下一章详细介绍。
- cid（types.ID 类型）：集群的 ID。

snapshotHandler.ServeHTTP()方法除了读取对端节点发来的快照数据，还会在本地生成相应的快照文件，并将快照数据通过 Raft 接口传递给底层的 etcd-raft 模块进行处理，其具体大致实现如下：

```
func (h *snapshotHandler) ServeHTTP(w http.ResponseWriter, r *http.Request) {
    // 首先进行一系列的检查，例如：检查请求的 Method 是否为 POST，与前面介绍的 pipelineHandler
    // 类似(略)

    dec := &messageDecoder{r: r.Body}
    // 限制每次从底层连接读取的字节数上线，默认是 64KB，因为快照数据可能非常大，为了防止读取超时，
    // 只能每次读取一部分数据到缓冲区中，最后将全部数据拼接起来，得到完整的快照数据
    m, err := dec.decodeLimit(uint64(1 << 63)) // 读取时的异常处理(略)
    // 检测读取到的消息类型，是否为 MsgSnap，若不是，则直接返回异常信息
    if m.Type != raftpb.MsgSnap {
        http.Error(w, "wrong raft message type", http.StatusBadRequest)
        return
    }
    // 使用 Snapshotter 将快照数据保存到本地文件中，省略相关的异常处理代码
    n, err := h.snapshotter.SaveDBFrom(r.Body, m.Snapshot.Metadata.Index)

    // 调用 Raft.Process()方法，将 MsgSnap 消息传递给底层的 etcd-raft 模块进行处理，
    // 省略相关的异常处理代码
    if err := h.r.Process(context.TODO(), m); err != nil ...

    w.WriteHeader(http.StatusNoContent) // 返回的状态码是 204
}
```

本章小结

本章介绍了 Go 语言中常用的几个网络组件的工作原理，重点介绍 etcd-rafthttp 模块的功能和实现原理。在本章开始，介绍了 Go 语言中的 http.Server、RoundTripper 的基本功能及工作原



仅供非商业用途或交流学习使用

理。之后介绍了 etcd 网络模块中的 Transporter 接口，以及其在 etcd-rafthttp 模块中的实现——Transport 结构体。接着深入分析 Transport 结构体涉及的多个组件的具体实现，例如，Peer、pipeline、streamWriter、streamReader 和 snapshotSender，这些组件共同构成了 etcd-rafthttp 模块完整的功能。最后深入介绍了 pipelineHandler、streamHandler 和 snapshotHandler 的具体实现，它们主要用来接收对端节点发来的 HTTP 请求。希望通过本章的阅读，读者可以了解 etcd-rafthttp 模块的工作原理。



仅供非商业用途或交流学习使用



第 5 章 WAL 日志与快照

etcd-raft 模块为了保证 Raft 核心协议实现的简洁，并没有直接提供 WAL 日志与快照相关的实现逻辑，而是将其实现独立到 etcd-wal 模块与 etcd-snap 模块，其中提供了操作 WAL 日志文件与快照文件的相关实现。上层模块自身调用 etcd-wal 模块与 etcd-snap 模块即可完成读写 WAL 日志文件与快照文件的相关操作。本章将详细介绍 etcd-wal 模块与 etcd-snap 模块提供的具体功能及实现。

5.1 WAL日志

WAL（Write-ahead logging）是 etcd 实现一致性的重要手段之一。下面简单回顾一下处理一条 Entry 记录的大致流程：

（1）当客户端向 etcd 集群发送了一次请求之后，请求中的封装 Entry 记录会先被交给 etcd-raft 模块进行处理，其中，etcd-raft 模块会先将 Entry 记录保存到 raftLog.unstable 中。

（2）etcd-raft 模块将该 Entry 记录封装到前面介绍的 Ready 实例中，返回给上层模块进行持久化。

（3）当上层模块收到待持久化的 Entry 记录之后，会先将其记录到 WAL 日志文件中，然后进行持久化操作，最后通知 etcd-raft 模块进行处理。

（4）此时 etcd-raft 模块就会将该 Entry 记录从 unstable 移动到 storage 中保存。

（5）待该 Entry 记录被复制到集群中的半数以上节点时，该 Entry 记录会被 Leader 节点确认为已提交（committed），并封装进 Ready 实例返回给上层模块。

（6）此时上层模块即可将该 Ready 实例中携带的待应用 Entry 记录应用到状态机中。



这就是处理一条 Entry 记录大致过程，如图 5-1 所示。

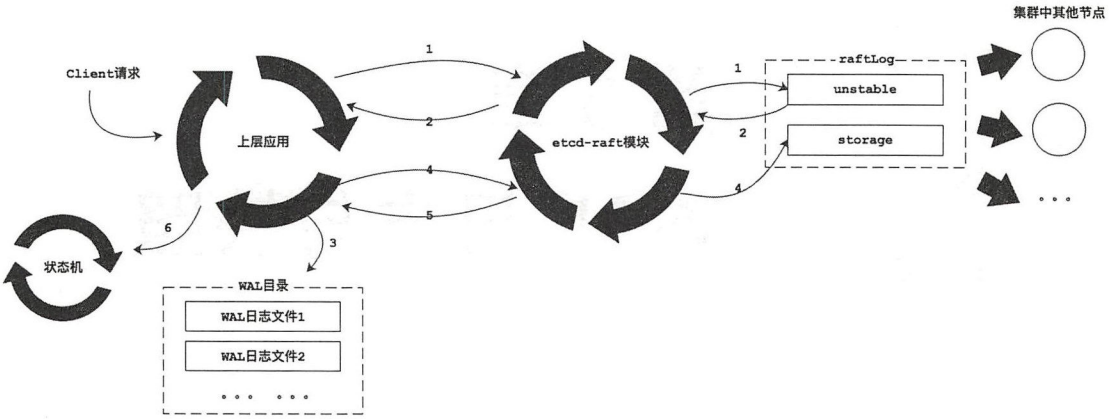


图 5-1

在 etcd-wal 模块中，首先需要介绍的就是结构体 WAL，它对外提供了 WAL 日志文件管理的核心 API。在操纵 WAL 日志时，对应的 WAL 实例有 read 和 append 两种模式，新创建的 WAL 实例处于 append 模式，该模式下只能向 WAL 中追加日志。当恢复一个节点时（例如，宕机节点的重启），就需要读取 WAL 日志的内容，此时刚打开的 WAL 实例处于 read 模式，它只能读取日志记录，当读取完全部的日志之后，WAL 实例转换成 append 模式，可以继续向其追加日志记录。

在 WAL 日志文件中，日志记录是通过 Record 表示的，该结构体通过 Protocol Buffers 生成，主要用于序列化和反序列化日志记录，其中各个字段的含义如下。

- Type 字段（int64 类型）：表示该 Record 实例的类型。
- Crc 字段（uint32 类型）：记录该 Record 实例的校验码。
- Data 字段（[]byte 类型）：记录真正的日志数据，根据日志的类型不同，Data 字段中保存的数据也有所相同。

Record 结构体还提供了一些简单的方法，后面遇到时会进行简单说明。根据 Record.Type 字段值，可以将日志记录分为如下几种类型。

- metadataType：该类型日志记录的 Data 字段中保存了一些元数据，在每个 WAL 文件的开头，都会记录一条 metadataType 类型的日志记录。
- entryType：该类型日志记录的 Data 字段中保存的是 Entry 记录，也就是客户端发送给服务端处理的数据，例如，raftexample 示例中客户端发送的键值对数据。
- stateType：该类型日志记录的 Data 字段中保存了当前集群的状态信息（即 HardState），在每次批量写入 entryType 类型日志记录之前，都会先写入一条 stateType 类型的日志



记录。

- crcType：该类型的日志记录主要用于数据校验。
- snapshotType：该类型的日志记录中保存了快照数据的相关信息（即 walpb.Snapshot，注意，其中不包含完整的快照数据）。

了解了这些基础知识之后，我们回到 WAL 结构体，先通过一张结构图了解 WAL 的大致结构，如图 5-2 所示。

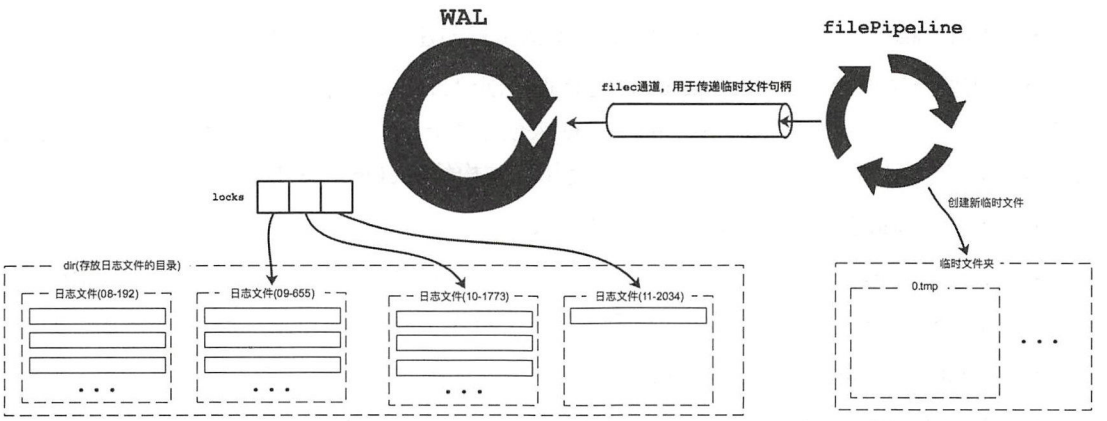


图 5-2

下面来看一下 WAL 结构体核心字段的含义。

- dir（string 类型）：存放 WAL 日志文件的目录路径。
- dirFile（*os.File 类型）：根据 dir 路径创建的 File 实例。
- metadata（[]byte 类型）：在每个 WAL 日志文件的头部，都会写入 metadata 元数据。
- state（raftpb.HardState 类型）：WAL 日志记录的追加是批量的，在每次批量写入 entryType 类型的日志之后，都会再追加一条 stateType 类型的日志记录，在 HardState 中记录了当前的 Term、当前节点的投票结果和已提交日志的位置。
- start（walpb.Snapshot 类型）：每次读取 WAL 日志时，并不会每次都从头开始读取，而是通过这里的 start 字段指定具体的起始位置。walpb.Snapshot 中的 Index 字段记录了对应快照数据所涵盖的最后一条 Entry 记录的索引值，Term 字段则记录了对应 Entry 记录的 Term 值。在读取 WAL 日志文件时，我们就可以根据这些信息，找到合适的位置并开始读取记录。
- decoder（*decoder 类型）：负责在读取 WAL 日志文件时，将二进制数据反序列化成 Record 实例。



- encoder（*encoder 类型）：负责将写入 WAL 日志文件的 Record 实例进行序列化成二进制数据。
- mu（sync.Mutex 类型）：读写 WAL 日志时需要加锁同步。
- enti（uint64 类型）：WAL 中最后一条 Entry 记录的索引值。
- locks（[]*fileutil.LockedFile 类型）：当前 WAL 实例管理的所有 WAL 日志文件对应的句柄。
- fp（*filePipeline 类型）：filePipeline 实例负责创建新的临时文件。

这里需要特别说明一下 filePipeline，它负责预创建日志文件并为日志文件预分配空间。在 filePipeline 中会启动一个独立的后台 goroutine 来创建“.tmp”结尾的临时文件，当进行日志文件切换时，直接将临时文件进行重命名即可使用。结构体 filePipeline 中各个字段的含义如下。

- dir（string 类型）：存放临时文件的目录。
- size（int64 类型）：创建临时文件时预分配空间的大小，默认是 64MB（由 wal.SegmentSizeBytes 指定，该值也是每个日志文件的大小）。
- count（int 类型）：当前 filePipeline 实例创建的临时文件数。
- filec（chan *fileutil.LockedFile 类型）：新建的临时文件句柄会通过 filec 通道返回给 WAL 实例使用。
- errc（chan error 类型）：当创建临时文件出现异常时，则将异常传递到 errc 通道中。
- donec（chan struct{}类型）：当 filePipeline.Close()被调用时会关闭 donec 通道，从而通知 filePipeline 实例删除最后一次创建的临时文件。

在 newFilePipeline()方法中，除了创建 filePipeline 实例，还会启动一个后台 goroutine 来执行 filePipeline.run()方法，该后台 goroutine 中会创建新的临时文件并将其句柄传递到 filec 通道中。filePipeline.run()方法的具体实现如下：

```
func (fp *filePipeline) run() {
  defer close(fp.errc)
  for {
    f, err := fp.alloc() // 调用 alloc()方法创建临时文件
    // 如果创建临时文件失败，则将异常传递到 errc 通道中(略)

    select {
    case fp.filec <- f: // 将上面创建的临时文件句柄传递到 filec 通道中
    case <-fp.donec: // 关闭时触发，删除最后一次创建的临时文件
```



```
            os.Remove(f.Name())
            f.Close()
            return
        }
    }
}
```

下面来看一下 filePipeline.alloc()方法是如何创建临时文件的，其具体实现如下：

```
func (fp *filePipeline) alloc() (f *fileutil.LockedFile, err error) {
    // 为了防止与前一个创建的临时文件重名，新建临时文件的编号是 0 或是 1
    fpath := filepath.Join(fp.dir, fmt.Sprintf("%d.tmp", fp.count%2))
    // 创建临时文件，注意文件的模式和权限
    if f, err = fileutil.LockFile(fpath, os.O_CREATE|os.O_WRONLY,
                    fileutil.PrivateFileMode); err != nil {
        return nil, err
    }
    // 尝试预分配空间，如果当前系统不支持预分配文件空间，则并不会报错
    if err = fileutil.Preallocate(f.File, fp.size, true); err != nil {
        f.Close() // 如果出现异常，则会关闭 donec 通道
        return nil, err
    }
    fp.count++ // 递增创建的文件数量
    return f, nil // 返回创建的临时文件
}
```

在 WAL 切换日志文件时会调用 filePipeline.Open()方法，从 filec 通道中获取之前创建好的临时文件，具体实现如下：

```
func (fp *filePipeline) Open() (f *fileutil.LockedFile, err error) {
    select {
    case f = <-fp.filec: // 从 filec 通道中获取已经创建好的临时文件，并返回
    case err = <-fp.errc: // 如果创建临时文件时的异常，则通过 errc 通道中获取，并返回
    }
    return
}
```



5.1.1 初始化

介绍完 filePipeline 结构体之后，我们回到 WAL 继续分析。先来看一下 wal.Create()方法，该方法不仅会创建 WAL 实例，而是做了很多初始化工作，其大致步骤如下：

（1）创建临时目录，并在临时目录中创建编号为“0-0”的 WAL 日志文件，WAL 日志文件名由两部分组成，一部分是 seq（单调递增），另一部分是该日志文件中的第一条日志记录的索引值。

（2）尝试为该 WAL 日志文件预分配磁盘空间。

（3）向该 WAL 日志文件中写入一条 crcType 类型的日志记录、一条 metadataType 类型的日志记录及一条 snapshotType 类型的日志记录。

（4）创建 WAL 实例关联的 filePipeline 实例。

（5）将临时目录重命名为 WAL.dir 字段指定的名称。

这里之所以先使用临时目录完成初始化操作再将其重命名的方式，主要是为了让整个初始化过程看上去是一个原子操作。wal.Create()方法的具体实现如下：

```
func Create(dirpath string, metadata []byte) (*WAL, error) {
  // 检测文件夹是否存在(略)
  tmpdirpath := filepath.Clean(dirpath) + ".tmp" // 得到临时目录的路径
  if fileutil.Exist(tmpdirpath) { // 清空临时目录中的文件
    if err := os.RemoveAll(tmpdirpath); err != nil {
      return nil, err
    }
  }

  if err := fileutil.CreateDirAll(tmpdirpath);... // 创建临时文件夹

  p := filepath.Join(tmpdirpath, walName(0, 0))// 第一个 WAL 日志文件的路径(文件名为 0-0)
  // 创建临时文件，注意文件的模式和权限，异常检查(略)
  f, err := fileutil.LockFile(p, os.O_WRONLY|os.O_CREATE, fileutil.PrivateFileMode)

  // 移动临时文件的 offset 到文件结尾处，
  // 注意 Seek()方法的第二个参数(0 是相对文件开头，1 是相对当前 offset，2 是相对文件结尾)
  if _, err = f.Seek(0, io.SeekEnd); err != nil {
    return nil, err
  }
  // 对新建的临时文件进行空间预分配，默认值是 64MB(SegmentSizeBytes)
```



```
    if err = fileutil.Preallocate(f.File, SegmentSizeBytes, true); err != nil {
        return nil, err
    }

    w := &WAL{ // 创建 WAL 实例
        dir:      dirpath,  // 存放 WAL 日志文件的目录的路径
        metadata: metadata, // 元数据
    }
    w.encoder, err = newFileEncoder(f.File, 0) // 创建写 WAL 日志文件的 encoder，异常检查(略)
    // 将 WAL 日志文件对应的 LockedFile 实例记录到 locks 字段中，表示当前 WAL 实例正在管理该日志文件
    w.locks = append(w.locks, f)

    // 创建一条 crcType 类型的日志写入 WAL 日志文件，异常检查(略)
    if err = w.saveCrc(0);...
    // 将元数据封装成一条 metadataType 类型的日志记录写入 WAL 日志文件，异常检查(略)
    if err = w.encoder.encode(&walpb.Record{Type: metadataType, Data: metadata});...
    // 创建一条空的 snapshotType 类型的日志记录写入临时文件，异常检查(略)
    if err = w.SaveSnapshot(walpb.Snapshot{});...

    // 将临时目录重命名，并创建 WAL 实例关联的 filePipline 实例，异常检查(略)
    if w, err = w.renameWal(tmpdirpath);...

    // 临时目录重命名之后，需要将重命名操作刷新到磁盘上，异常检查(略)
    pdir, perr := fileutil.OpenDir(filepath.Dir(w.dir))
    if perr = fileutil.Fsync(pdir); ... // 同步磁盘的操作，异常检查(略)
    return w, nil // 返回 WAL 实例
}
```

WAL.renameWal()方法在不同系统平台的操作略有不同，但是主要操作都是重命名临时目录并创建关联的 filePipeline 实例。下面是针对 Linux 系统的具体实现（该实现在 wal_unix.go 中，Window 系统对应的实现在 wal_windows.go 中）：

```
func (w *WAL) renameWal(tmpdirpath string) (*WAL, error) {
    if err := os.RemoveAll(w.dir); ... // 清空 wal 文件夹，异常检查(略)
    if err := os.Rename(tmpdirpath, w.dir);... // 重命名临时文件夹，异常检查(略)

    w.fp = newFilePipeline(w.dir, SegmentSizeBytes) // 创建 WAL 实例关联的 filePipeline
                                                    // 实例
```



```
    df, err := fileutil.OpenDir(w.dir)
    w.dirFile = df // WAL.dirFile 字段记录了 WAL 日志目录对应的文件句柄
    return w, err
}
```

5.1.2 打开日志

在开始介绍 wal.Open*()方法之前，需要读者注意两个地方：

- 在打开 WAL 日志文件时，可以指定此次打开日志文件的模式是只读模式还是读写模式。
- 在打开 WAL 日志文件时，都会指定随后的读取操作的起始 index，而不是每次都从第一个日志文件开始读取。

wal 模块根据上面两点，提供了 Open()和 OpenForRead()两个函数，两者的区别在于：使用 Open()函数创建的 WAL 实例读取完全部日志后，可以继续追加日志；而 OpenForRead()函数创建的 WAL 实例只能用于读取日志，不能追加日志。这两个方法底层都是通过调用 openAtIndex()方法完成的，该方法的具体实现如下：

```
// 需要注意 snap 和 write 参数，snap.Index 指定了日志读取的起始位置，
// write 参数则指定了打开日志文件的模式
func openAtIndex(dirpath string, snap walpb.Snapshot, write bool) (*WAL, error) {
    // 获取全部的 WAL 日志文件名，并且这些文件名会进行排序，readWalNames()方法的实现比较简单，
    // 这里不再赘述，感兴趣的读者可以分析一下。此处省略异常处理的相关代码
    names, err := readWalNames(dirpath)

    // 根据 WAL 日志文件名的规则，查找上面得到的所有文件名，找到 index 最大且 index 小于 snap.Index
    // 的 WAL 日志文件，并返回该文件在 names 数组中的索引(nameIndex)，searchIndex()方法的实现比
    // 较简单，感兴趣的读者可以分析一下。此处省略异常处理的相关代码
    nameIndex, ok := searchIndex(names, snap.Index)

    rcs := make([]io.ReadCloser, 0)
    rs := make([]io.Reader, 0) // 注意该切片中元素的类型，在后面介绍的 decoder 中会使用到
    ls := make([]*fileutil.LockedFile, 0)
    // 从 nameIndex 开始读取剩余的 WAL 日志文件
    for _, name := range names[nameIndex:] {
        p := filepath.Join(dirpath, name) // 获取 WAL 日志的绝对路径
        if write { // 以读写模式打开 WAL 日志文件
```



```
            // 打开 WAL 日志文件并且对文件加锁，注意打开文件的模式，省略异常处理的相关代码
            l, err := fileutil.TryLockFile(p, os.O_RDWR, fileutil.PrivateFileMode)
            ls = append(ls, l) // 文件句柄记录到 ls 和 rcs 这两个切片中
            rcs = append(rcs, l)
        } else {
            // 如果 write 参数为 false，则表示以只读模式打开文件，省略异常处理的相关代码
            rf, err := os.OpenFile(p, os.O_RDONLY, fileutil.PrivateFileMode)
            ls = append(ls, nil) // 注意，这里只将文件句柄添加到了 rcs 切片中
            rcs = append(rcs, rf)
        }
        rs = append(rs, rcs[len(rcs)-1]) // 将文件句柄记录到 rs 切片中
    }
    closer := func() error { return closeAll(rcs...) }  // 后面关闭文件时，会调用该函数
    w := &WAL{ // 创建 WAL 实例
        dir:      dirpath,
        start:    snap, // 记录 SnapShot 的信息
        // 创建用于读取日志记录的 decoder 实例，这里并没有初始化 encoder，所以还不能写入日志记录
        decoder:  newDecoder(rs...),
        // 如果是只读模式，在读取完全部日志文件之后，则会调用该方法关闭所有日志文件
        readClose: closer,
        locks:    ls, // 当前 WAL 实例管理的日志文件
    }

    // 如果是读写模式，读取完全部日志文件之后，由于后续有追加操作，所以不需要关闭日志文件；另外，还
    // 要为 WAL 实例创建关联的 filePipeline 实例，用于产生新的日志文件
    if write {
        w.readClose = nil
        w.fp = newFilePipeline(w.dir, SegmentSizeBytes) // 创建 filePipeline
    }
    return w, nil
}
```

5.1.3　读取日志

通过 Open()函数或 OpenForRead()函数创建 WAL 实例之后，就可以调用其 ReadAll()方法读取日志了。WAL.ReadAll()方法首先从 WAL.start 字段指定的位置开始读取日志记录，读取完毕



仅供非商业用途或交流学习使用

之后，会根据读取的情况进行一系列异常处理。然后根据当前 WAL 实例的模式进行不同的处理：如果处于读写模式，则需要先对后续的 WAL 日志文件进行填充并初始化 WAL.encoder 字段，为后面写入日志做准备；如果处于只读模式下，则需要关闭所有的日志文件。另外需要注意的是，WAL.ReadAll()方法的几个返回值都是从日志记录中读取到的。WAL.ReadAll()方的具体实现如下：

```
func (w *WAL) ReadAll() (metadata []byte, state raftpb.HardState,
                         ents []raftpb.Entry, err error) {
  // 省略加锁和解锁的相关代码
  rec := &walpb.Record{} // 创建 Record 实例
  decoder := w.decoder // 解码器，负责读取日志文件，并将日志数据反序列化成 Record 实例

  var match bool // 标识是否找到了 start 字段对应的日志记录
  // 循环读取 WAL 日志文件中的数据，多个 WAL 日志文件的切换是在 decoder 中完成的，后面会详细分析
  // 其实现
  for err = decoder.decode(rec); err == nil; err = decoder.decode(rec) {
    switch rec.Type { // 根据日志记录的类型进行分类处理
    case entryType: // 读取到 entryType 类型的日志
      e := mustUnmarshalEntry(rec.Data) // 反序列化 Record.Data 中记录的数据，得到 Entry 实例
      if e.Index > w.start.Index { // 将 start 之后的 Entry 记录添加到 ents 中保存
        ents = append(ents[:e.Index-w.start.Index-1], e)
      }
      w.enti = e.Index // 记录读取到的最后一条 Entry 记录的索引值
    case stateType: // 读取到 stateType 类型的日志记录
      state = mustUnmarshalState(rec.Data) // 更新待返回的 HardState 状态信息
    case metadataType: // 读取到 metadataType 类型的日志记录
      // 检测 metadata 数据是否发生冲突，如果冲突，则抛出异常
      metadata = rec.Data // 更新待返回的元数据
    case crcType: // 读取到 crcType 类型的日志记录，更新 decoder 中的 crc 信息
      decoder.updateCRC(rec.Crc) // 更新 decodr.crc 字段，其他操作(略)
    case snapshotType: // 读取到 snapshotType 类型的日志记录
      var snap walpb.Snapshot
      pbutil.MustUnmarshal(&snap, rec.Data) // 解析快照相关的数据
      if snap.Index == w.start.Index { // 异常检查(略)
        match = true // 更新 match
      }
    default: // 其他未知类型的日志记录，返回异常(略)
    }
```



仅供非商业用途或交流学习使用


```
    }
    // 到这里，读取 WAL 日志文件的操作就完成了(当然，中间可能出现了异常)
    switch w.tail() { // 根据 WAL.locks 字段是否有值判断当前 WAL 是什么模式
    case nil:
        // 对于只读模式，并不需要将全部的日志都读出来，因为以只读模式打开 WAL 日志文件时，
        // 并没有加锁，所以最后一条日志记录可能只写了一半，从而导致 io.ErrUnexpectedEOF 异常
        if err != io.EOF && err != io.ErrUnexpectedEOF {
            state.Reset()
            return nil, state, nil, err
        }
    default:
        // 对于读写模式，则需要将日志记录全部读出来，所以此处不是 EOF 异常，则报错(略)
        // 将文件指针移动到读取结束的位置，并将文件后续部分全部填充为 0
        if _, err = w.tail().Seek(w.decoder.lastOffset(), io.SeekStart); ...
                                                        // 异常处理(略)
        if err = fileutil.ZeroToEnd(w.tail().File); ...    // 异常处理(略)
    }
    if !match { // 如果在读取过程中没有找到与 start 对应的日志记录，则抛出异常
        err = ErrSnapshotNotFound
    }

    if w.readClose != nil { // 如果是只读模式，则关闭所有日志文件
        w.readClose() // WAL.readClose 实际指向的是 WAL.CloseAll()方法
        w.readClose = nil
    }
    w.start = walpb.Snapshot{} // 清空 start 字段
    w.metadata = metadata
    if w.tail() != nil { // 如果是读写模式，则初始化 WAL.encoder 字段，为后面写入日志做准备
        w.encoder, err = newFileEncoder(w.tail().File, w.decoder.lastCRC())
    }
    w.decoder = nil // 清空 WAL.decoder 字段，后续不能再用该 WAL 实例进行读取了
    return metadata, state, ents, err
}
```

实际读取 WAL 日志的功能是在 decoder.decode()的方法中完成的，下面简单介绍一下 decoder 的实现，先来看一下 decoder 的核心字段，如下所示。

- mu (sync.Mutex 类型): 在 decoder 开始读取日志文件时，需要加锁同步。



仅供非商业用途或交流学习使用

- brs（[]*bufio.Reader 类型）：该 decoder 实例通过该字段中记录的 Reader 实例读取相应的日志文件，这些日志文件就是在前面介绍的 wal.openAtIndex()方法中打开的日志文件。
- lastValidOff（int64 类型）：读取日志记录的指针。
- crc（hash.Hash32 类型）：校验码。

decoder.decode()方法底层是通过调用 decoder.decodeRecord()方法实现的，该方法的具体实现如下：

```
func (d *decoder) decodeRecord(rec *walpb.Record) error {
  // 检测 brs 字段长度，决定是否还有日志文件需要读取(略)

  l, err := readInt64(d.brs[0]) // 读取第一个日志文件中的第一个日志记录的长度

  // 是否读到文件尾，或是读取到了预分配的部分，这都表示读取操作结束
  if err == io.EOF || (err == nil && l == 0) {
    d.brs = d.brs[1:] // 更新 brs 字段，将其中第一个日志文件对应的 Reader 清除掉
    if len(d.brs) == 0 { // 如果后面没有其他日志文件可读则返回 EOF 异常，表示读取正常结束
      return io.EOF
    }
    d.lastValidOff = 0 // 若后续还有其他日志文件待读取，则需要切换文件，这里会重置 lastValidOff
    return d.decodeRecord(rec) // 递归调用 decodeRecord()方法
  }
  // 计算当前日志记录的实际长度及填充数据的长度，并创建相应的 data 切片
  recBytes, padBytes := decodeFrameSize(l)
  data := make([]byte, recBytes+padBytes)
  if _, err = io.ReadFull(d.brs[0], data); err != nil { // 从日志文件中读取指定长度的
    // 字节数如果读取不到指定的字节数，则会返回 EOF 异常，此时认为读取到了写入一半的日志记录，
    // 需要返回 ErrUnexpectedEOF 异常
    if err == io.EOF {
      err = io.ErrUnexpectedEOF
    }
    return err
  }
  // 将 0~recBytes 反序列化成 Record
  if err := rec.Unmarshal(data[:recBytes]);... // 异常检查(略)
  // 进行 crc 校验(略)
  d.lastValidOff += recBytes + padBytes + 8 // 将 lastValidOff 后移，准备读取下一条日志记录
```



仅供非商业用途或交流学习使用


```
    return nil
}
```

5.1.4　追加日志

正如上一小节介绍，通过 WAL.ReadAll()方法读取完全部日志记录之后，WAL.encoder 字段才会被初始化，自此之后，我们才能通过该 WAL 实例向日志文件中追加日志记录。WAL 对外提供了追加日志的方法，分别是 Save()方法和 SaveSnapshot()方法。

WAL.Save()方法先将待写入的 Entry 记录封装成 entryType 类型的 Record 实例，然后将其序列化并追加到日志段文件中，之后将 HardState 封装成 stateType 类型的 Record 实例，并序列化写入日志段文件中，最后将这些日志记录同步刷新到磁盘。WAL.Save()方法的具体实现如下：

```
func (w *WAL) Save(st raftpb.HardState, ents []raftpb.Entry) error {
    w.mu.Lock() // 加锁同步
    defer w.mu.Unlock() // 日志记录追加完成之后，释放锁
    // 边界检查，如果待写入的 HardState 和 Entry 数组都为空，则直接返回(略)；否则就需要将修改同步
    // 到磁盘上
    mustSync := raft.MustSync(st, w.state, len(ents))

    // 遍历待写入的 Entry 数组，将每个 Entry 实例序列化并封装 entryType 类型的日志记录，写入日志文件
    for i := range ents {
        if err := w.saveEntry(&ents[i]); ... // 如果发生异常，则将异常返回(略)
    }

    // 将状态信息(HardState)序列化并封装成 stateType 类型的日志记录，写入日志文件
    if err := w.saveState(&st); ...

    // 获取当前日志段文件的文件指针的位置，异常处理(略)
    curOff, err := w.tail().Seek(0, io.SeekCurrent)
    // 如果未写满预分配的空间，将新日志刷新到磁盘后，即可返回
    if curOff < SegmentSizeBytes {
        if mustSync {
            return w.sync() // 将上述追加的日志记录同步刷新到磁盘上
        }
        return nil
    }
```



仅供非商业用途或交流学习使用


```
    return w.cut() // 当前文件大小已超出了预分配的空间，则需要进行日志文件的切换
}
```

在 WAL.Save()方法中追加日志记录的操作分别由 saveEntry()方法和 saveState()方法完成，两者的实现非常相似，saveState()方法留给读者自己进行分析，这里简单介绍 saveEntry()方法的实现，如下所示。

```
func (w *WAL) saveEntry(e *raftpb.Entry) error {
    b := pbutil.MustMarshal(e) // 将 Entry 记录序列化
    // 将序列化后的数据封装成 entryType 类型的 Record 记录
    rec := &walpb.Record{Type: entryType, Data: b}
    // 通过 encoder.encode()方法追加日志记录，省略相关的异常处理代码
    if err := w.encoder.encode(rec); ...
    w.enti = e.Index // 更新 WAL.enti 字段，其中保存了最后一条 Entry 记录的索引值
    return nil
}
```

WAL.Save()在完成日志记录的追加之后，会调用 sync()方法将日志同步刷新到磁盘上，其具体实现如下：

```
func (w *WAL) sync() error {
    if w.encoder != nil {
        // 先使用 encoder.flush()方法进行同步刷新，encoder 的具体实现后面详细介绍
        if err := w.encoder.flush();... // 异常处理(略)
    }
    start := time.Now()
    // 使用操作系统的 fdatasync 将数据真正刷新到磁盘上
    err := fileutil.Fdatasync(w.tail().File)
    // 这里会对该刷新操作的执行时间进行监控，如果刷新操作执行的时间长于指定的时间(默认值是 1s)，
    // 则输出警告日志(略)
    return err
}
```

通过本节的介绍可知，encoder.encode()方法是真正完成日志写入功能的地方，这里简单介绍一下 encoder 的实现，先来看一下 encoder 的核心字段。

- mu（sync.Mutex 类型）：在进行读写文件的时候，需要加锁同步。
- bw（*ioutil.PageWriter 类型）：PageWriter 是带有缓冲区的 Writer，在写入时，每写满



仅供非商业用途或交流学习使用

一个 Page 大小的缓冲区，就会自动触发一次 Flush 操作，将数据同步刷新到磁盘上。每个 Page 的大小是由 walPageBytes 常量指定的。

- buf ([]byte 类型): 日志序列化之后，会暂存在该缓冲区中，该缓冲区会被复用，这就防止了每次序列化创建缓冲区带来的开销。
- uint64buf ([]byte 类型): 在写入一条日志记录时，该缓冲区用来暂存一个 Frame 的长度的数据（Frame 由日志数据和填充数据构成）。

在 encoder.encode()方法中，会对日志记录进行序列化，并完成 8 字节对齐，具体实现如下所示。

```
func (e *encoder) encode(rec *walpb.Record) error {
  e.mu.Lock() // 加锁同步
  defer e.mu.Unlock()// 整个方法结束后，释放锁
  // 计算 crc 校验码(略)

  // 将待写入到日志记录进行序列化，如果
  if rec.Size() > len(e.buf) { // 日志记录太大，无法复用 encoder.buf 这个缓冲区，则直接序列化
    data, err = rec.Marshal() // 省略相关的错误处理代码
  } else { // 复用 encoder.buf 这个缓冲区，省略相关的错误处理代码
    n, err = rec.MarshalTo(e.buf)
    data = e.buf[:n]
  }

  // 计算序列化之后的数据长度，在 encodeFrameSize()方法中会完成 8 字节对齐，
  // 这里将真正的数据和填充数据看作一个 Frame，返回值分别是整个 Frame 的长度，
  // 以及其中填充数据的长度
  lenField, padBytes := encodeFrameSize(len(data))
  // 将 Frame 的长度序列化到 encoder.uint64buf 数组中，然后写入文件，异常处理(略)
  if err = writeUint64(e.bw, lenField, e.uint64buf); ...

  if padBytes != 0 {
    data = append(data, make([]byte, padBytes)...) // 向 data 中写入填充字节
  }
  _, err = e.bw.Write(data) // 将 data 中的序列化数据写入文件
  return err
}
```



仅供非商业用途或交流学习使用

5.1.5 文件切换

随着 WAL 日志文件的不断写入，单个日志文件会不断变大。在前面提到过，每个日志文件的大小是有上限的，该阈值由 SegmentSizeBytes 指定（默认值是 64MB），该值也是日志文件预分配磁盘空间的大小。当单个日志文件的大小超过该值时，就会触发日志文件的切换，该切换过程是在 WAL.cut()方法中实现的，在上一小节读 WAL.Save()方法的分析中，也看到了该方法的身影。

WAL.cut()方法首先通过 filePipeline 获取一个新建的临时文件，然后写入 crcType 类型、metaType 类型、stateType 类型等必要日志记录（这个步骤与前面介绍的 Create()方法类似），然后将临时文件重命名成符合 WAL 日志命名规范的新日志文件，并创建对应的 encoder 实例更新到 WAL.encoder 字段。WAL.cut()方法的具体实现如下：

```
func (w *WAL) cut() error {
    // 获取当前日志文件的文件指针位置，异常检查(略)
    off, serr := w.tail().Seek(0, io.SeekCurrent)

    // 根据当前的文件指针位置，将后续填充内容 Truncate 掉，这主要是处理提早切换和
    // 预分配空间未使用的情况，truncate 后可以释放该日志文件后续未使用的空间，异常处理(略)
    if err := w.tail().Truncate(off);...

    // 紧接着执行一次 WAL.sync()方法，将修改同步刷新到磁盘上，异常处理(略)
    if err := w.sync();...

    // 根据当前最后一个日志文件的名称，确定下一个新日志文件的名称，
    // seq()方法返回当前最后一个日志文件的编号，w.enti 记录了当前最后一条日志记录的索引值
    fpath := filepath.Join(w.dir, walName(w.seq()+1, w.enti+1))

    // 从 filePipeline 获取新建的临时文件，filePipeline.Open()方法在前面介绍过了，这里不再赘述，
    // 这里省略异常处理的相关代码
    newTail, err := w.fp.Open()
    w.locks = append(w.locks, newTail) // 将临时文件的句柄保存到 WAL.locks 中
    prevCrc := w.encoder.crc.Sum32()
    // 创建临时文件对应的 encoder 实例，并更新到 WAL.encoder 字段中，省略异常处理的相关代码
    w.encoder, err = newFileEncoder(w.tail().File, prevCrc)

    // 向临时文件中追加一条 crcType 类型的日志记录，省略异常处理的相关代码
```



仅供非商业用途或交流学习使用


```
    if err = w.saveCrc(prevCrc);...
    // 向临时文件中追加一条 metadataType 类型的日志记录，省略异常处理的相关代码
    if err = w.encoder.encode(&walpb.Record{Type: metadataType, Data: w.metadata}); ...
    // 向临时文件中追加一条 stateType 类型的日志记录，省略异常处理的相关代码
    if err = w.saveState(&w.state); ...
    // 通过 WAL.sync()方法将上述修改同步刷新到磁盘上，省略异常处理的相关代码
    if err = w.sync();...

    // 记录当前文件指针的位置，为重命名之后，重新打开文件做准备，省略异常处理的相关代码
    off, err = w.tail().Seek(0, io.SeekCurrent)

    // 将临时文件重命名成之前得到的新日志文件名称，省略异常处理的相关代码
    if err = os.Rename(newTail.Name(), fpath);...

    // 将重命名这一操作同步刷新到磁盘上，fsync 操作不仅会将文件数据刷新到磁盘上，还
    // 会将文件的元数据也刷新到磁盘上(例如，文件的长度和名称等)，省略异常处理的相关代码
    if err = fileutil.Fsync(w.dirFile); ...

    newTail.Close() // 关闭临时文件对应的句柄

    // 打开重命名后的新日志文件，省略异常处理的相关代码
    if newTail, err = fileutil.LockFile(fpath, os.O_WRONLY, fileutil.PrivateFileMode);...
    // 将文件指针的位置移动到之前保存的位置，省略异常处理的相关代码
    if _, err = newTail.Seek(off, io.SeekStart); ...

    w.locks[len(w.locks)-1] = newTail // 将 WAL.locks 中最后一项更新成新日志文件

    prevCrc = w.encoder.crc.Sum32()
    // 创建新日志文件对应的 encoder 实例，并更新到 WAL.encoder 字段中，异常检查(略)
    w.encoder, err = newFileEncoder(w.tail().File, prevCrc)
    return nil
}
```

到这里为止，etcd-wal 模块的核心实现就已经全部介绍完了。在下一节中，我们将开始介绍快照存储相关的内容。



仅供非商业用途或交流学习使用

5.2 SnapShotter

随着节点的运行，会处理客户端和集群中其他节点发来的大量请求，相应的 WAL 日志量会不断增加，会产生大量的 WAL 日志文件，另外 etcd-raft 模块中的 raftLog 中也会存储大量的 Entry 记录，这就会导致资源浪费。当节点宕机之后，如果要恢复其状态，则需要从头读取全部的 WAL 日志文件，这显然是非常耗时的。为了解决这些问题，etcd 会定期创建快照并将其保存到本地磁盘中，在恢复节点状态时会先加载快照文件，使用该快照数据将节点恢复到对应的状态，之后从快照数据之后的相应位置开始读取 WAL 日志文件，最终将节点恢复到正确的状态。

与 WAL 日志的管理类似，etcd 并未在 etcd-raft 模块中实现将快照管理，而是将其独立到了 snap 模块中。其中 SnapShotter 通过文件的方式管理快照数据，它是 snapshot 模块的核心。在 SnapShotter 结构体中只有一个 dir 字段（string 类型），该字段指定了存储快照文件的目录位置。

Snapshotter.SaveSnap()方法的主要功能就是将快照数据保存到快照文件中，其底层是通过调用 save()方法实现的。save()方法的具体实现如下：

```
func (s *Snapshotter) save(snapshot *raftpb.Snapshot) error {
  start := time.Now()
  // 创建快照文件名，快照文件的名称由三部分组成，分别是快照所涵盖的最后一条 Entry 记录的 Term、
  // Index 和“.snap”后缀
  fname := fmt.Sprintf("%016x-%016x%s", snapshot.Metadata.Term,
       snapshot.Metadata.Index, snapSuffix)
  b := pbutil.MustMarshal(snapshot) // 将快照数据进行序列化
  // 计算 crc32 校验码(略)

  // 将序列化后的数据和校验码封装成 snappb.Snapshot 实例，
  // 这里简单了解一下 raftpb.Snapshot 和 snappb.Snapshot 的区别，前者包含了 SnapShot 数据
  // 及一些元数据（例如，该快照数据所涵盖的最后一条 Entry 记录的 Term 和 Index）；后者则是在前者序
  // 列化之后的封装，其中还记录了相应的校验码等信息
  snap := snappb.Snapshot{Crc: crc, Data: b}
  d, err := snap.Marshal() // 将 snappb.Snapshot 实例序列化
  // 省略异常处理及记录监控信息的相关代码

  // 将 snappb.Snapshot 序列化后的数据写入文件，并同步刷新到磁盘，省略异常处理相关的代码
  err = pioutil.WriteAndSyncFile(filepath.Join(s.dir, fname), d, 0666)
  return err
}
```

仅供非商业用途或交流学习使用

下面再来看一下 Snapshotter.Load()方法，该方法会读取指定目录下的全部快照文件，并查找其中最近的可用快照文件，然后通过 Snapshotter.loadSnap()方法加载其中的快照数据，其具体实现如下：

```
func (s *Snapshotter) Load() (*raftpb.Snapshot, error) {
    // 获取全部快照文件的名称，其中会检查文件的后缀名，过滤不合法的文件，省略异常处理的相关代码
    // 另外，返回的快照文件的名称是经过排序的
    names, err := s.snapNames()

    var snap *raftpb.Snapshot
    for _, name := range names {
        // 按序读取快照文件，直到读取到第一个合法的快照文件，对于读取失败的快照文件则添加“.broken”
        // 后缀，这样在下次调用 snapNames()方法获取可用快照文件时，就会将其过滤掉
        if snap, err = loadSnap(s.dir, name); err == nil {
            break
        }
    }
    return snap, nil
}
```

真正加载快照数据的功能是在 Snapshotter.loadSnap()方法中完成的，其具体实现如下：

```
func loadSnap(dir, name string) (*raftpb.Snapshot, error) {
    fpath := filepath.Join(dir, name) // 获取快照文件的绝对路径
    snap, err := Read(fpath) // 读取快照文件的内容
    if err != nil { // 如果读取失败，则将该快照文件的后缀名修改成“.broken”
        renameBroken(fpath)
    }
    return snap, err
}

// 下面是 Read()方法的具体实现
func Read(snapname string) (*raftpb.Snapshot, error) {
    b, err := ioutil.ReadFile(snapname) // 读取文件，省略异常处理的相关代码

    var serializedSnap snappb.Snapshot
    if err = serializedSnap.Unmarshal(b); ...   // 将读取到的数据进行反序列化，省略相关的异
                                                // 常处理
```

仅供非商业用途或交流学习使用


```
    // 使用 crc32 校验码验证数据是否正确(略)
    var snap raftpb.Snapshot
    // 反序列化 snappb.Snapshot 实例的 Data 字段，得到 raftpb.Snapshot 实例，省略相关的异常处理
    if err = snap.Unmarshal(serializedSnap.Data);...
    return &snap, nil
}
```

另外，Snapshotter.SaveDBFrom()方法和 DBFilePath()方法提供了另一套读写快照数据的方式，生成的快照文件名称也与上述方式不同。下面先来看一下 SaveDBFrom()方法的具体实现：

```
func (s *Snapshotter) SaveDBFrom(r io.Reader, id uint64) (int64, error) {
    f, err := ioutil.TempFile(s.dir, "tmp") // 创建临时文件，省略相关的异常处理
    var n int64
    n, err = io.Copy(f, r) // 将快照数据写入临时文件中
    if err == nil {
        err = fileutil.Fsync(f) // 将临时文件的改动刷新到磁盘
    }
    f.Close() // 关闭临时文件
    if err != nil {
        os.Remove(f.Name()) // 如果上述过程中出现异常，则删除临时文件
        return n, err
    }
    fn := s.dbFilePath(id) // 获取指定的“snap.db”文件，如果存在则将其删除
    if fileutil.Exist(fn) {
        os.Remove(f.Name())
        return n, nil
    }
    err = os.Rename(f.Name(), fn) // 重命名临时文件，省略相关的异常处理
    return n, nil
}

// 在 dbFilePath()方法中，我们可以清楚看到该方式生成的快照文件的名称结构
func (s *Snapshotter) dbFilePath(id uint64) string {
    return filepath.Join(s.dir, fmt.Sprintf("%016x.snap.db", id))
}
```

Snapshotter.DBFilePath()方法是在 dbFilePath()方法之上实现的，用于查找指定的快照 db 文

仅供非商业用途或交流学习使用

件，具体实现如下所示。

```
func (s *Snapshotter) DBFilePath(id uint64) (string, error) {
    // 尝试读取快照目录，主要是检测快照文件是否存在，省略相关的异常处理
    if _, err := fileutil.ReadDir(s.dir); err != nil ...
    // 获取指定的快照 db 文件的绝对路径
    if fn := s.dbFilePath(id); fileutil.Exist(fn) {
        return fn, nil
    }
    return "", ErrNoDBSnapshot
}
```

本章小结

在本章中，我们主要介绍了 etcd 中 wal 模块和 snap 模块的主要功能和具体实现。在本章的开始，我们介绍了 WAL 日志的相关内容，首先是记录 WAL 日志的意义，然后详细分析了 WAL 的结构及其主要功能的实现，例如，WAL 日志文件的模式、读写 WAL 日志文件、切换日志文件等功能，最后介绍了 snap 模块是如何实现快照数据的存储和加载。

希望通过本章的阅读，读者可以了解 WAL 日志和快照的基本工作原理，也帮助读者更清晰地理解前面介绍的 etcd-raft 模块和后续介绍的 etcd-server 模块的内容。

仅供非商业用途或交流学习使用



第 6 章 storage

随着 etcd 功能的不断完善及开源社区的不断壮大，etcd 于 2016 年 6 月发布了 v3.0.0 版本，从此进入了 etcd v3 时代。在 etcd v2 和 etcd v3 中，使用的 etcd-raft 模块变化不大，但使用的后端存储有很多不同之处。

在 etcd v2 和 v3 两个版本中，都使用了前面介绍的 etcd-raft 模块实现 Raft 协议相关的功能，但是它们底层使用的存储不同，对外提供的接口也不同。更重要的是，这两个版本的后端存储并不具有兼容性，实现也是相互独立的。也就是说，v3 版本的后端存储并不是由 v2 版本发展、升级而来的。另外，这两个版本的后端存储中的数据也是互相隔离的，这样在启动一个 etcd 节点之后，通过 v2 版本接口写入的数据只存在于 v2 版本的后端存储中，也只能通过 v2 版本的接口进行访问；而通过 v3 版本接口写入的数据只存在于 v3 版本的后端存储中，当然也只能通过 v3 版本的接口访问这些数据。

6.1 etcd v2 版本存储

etcd v2 版本的存储是一版完全基于内存的存储，它并没有将数据实时地写入到磁盘。在其需要进行持久化时，会将整个存储的数据序列化成 JSON 格式的数据并写入磁盘文件。在 etcd v2 版本的存储中，数据是以树形结构保存在内存中的，图 6-1 展示了 etcd v2 版本存储的大致结构。

在 etcd v2 版本的存储中，有一个全局的 currentIndex，每次出现数据变更的时候，该 currentIndex 会加 1，每个 event 都会关联到 currentIndex。当客户端调用 watch 接口（参数中增加 wait 参数）时，如果请求参数中有 waitIndex，并且 waitIndex 小于 currentIndex，则从 EventHistroy 表中查询 index 小于或等于 waitIndex，并且和 watch key 匹配的 event。如果有数据，则直接返

仅供非商业用途或交流学习使用

回。如果 EventHistroy 表中没有或者请求没有带 waitIndex，则放入 WatchHub 中，每个 key 会关联一个 watcher 列表。当有变更操作时，变更生成的 event 会放入 EventHistroy 表中，同时通知和该 key 相关的 watcher。

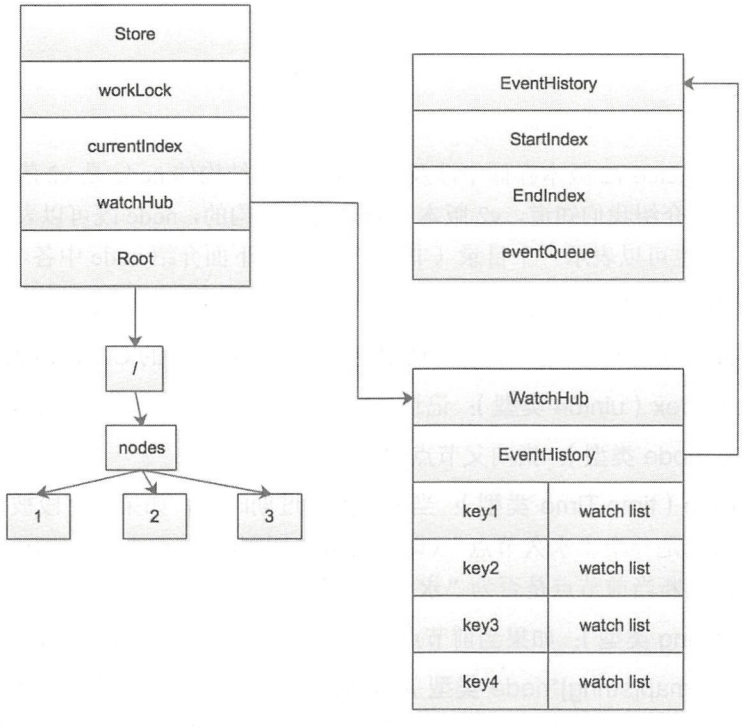


图 6-1

这里有几个影响使用的细节问题：

- EventHistroy 表是有长度限制的，默认情况下，其最大长度是 1000。也就是说，如果客户端停了较长时间，然后重新“watch”的时候，可能和该 waitIndex 相关的 event 实例已经被淘汰了，这种情况下就会丢失变更。
- 如果通知 watcher 的时候，出现了阻塞（每个 watcher 关联的 channel 有 100 个缓冲空间），则 etcd 会直接把 watcher 删除，会导致 wait 请求的连接中断，客户端需要重新连接。
- 为了防止内存被撑爆，etcd v2 版本存储的绝大多数 node 中都会设置过期时间，并通过定时清理的机制进行清理。

从而可以看出 etcd v2 的一些限制：

- 过期时间只能设置到每个 key 上，因此多个 key 要保证生命周期一致则比较困难。

- watcher 只能监听某一个 key 及其子节点（通过参数 recursive）。
- 很难通过 watcher 机制来实现完整的数据同步（因为有丢失变更的风险），所以当前的大多数使用方式是通过 watcher 得知变更，然后通过 get 重新获取数据的，并不完全依赖于 watcher 机制。

6.1.1 node

本小节主要介绍 etcd v2 版本存储中涉及的基本概念。结构体 node 是 v2 版本存储中最基本的元素，通过前面的介绍我们知道，v2 版本存储是树形结构的，node 既可以表示其中的一个键值对（叶子节点），也可以表示一个目录（非叶子节点）。下面介绍 node 中各个字段的含义。

- Path（string 类型）：当前节点的路径，格式类似“/dir1/dir2/key”。
- CreatedIndex（uint64 类型）：记录创建当前节点时对应的 CurrentIndex 值。
- ModifiedIndex（uint64 类型）：记录最后一次更新当前节点时对应的 CurrentIndex 值。
- Parent（*node 类型）：指向父节点的指针。
- ExpireTime（time.Time 类型）：当前节点的过期时间，如果该字段被设置为 0，则表示当前节点是一个“永久节点”（即不会被过期删除）。node.IsPermanent()方法通过检查该字段判断当前节点是否为“永久节点”。
- Value（string 类型）：如果当前节点表示一个键值对，则该字段记录了对应的值。
- Children（map[string]*node 类型）：如果当前节点表示一个目录节点，则该字段记录了其子节点。node.IsDir()方法通过检查该字段判断当前节点是否为目录节点。
- store（*store 类型）：记录当前节点关联的 v2 版本存储实例。

对于键值对和目录两种不同类型的节点，v2 版本存储分别提供了 newKV()和 newDir()两个不同的函数进行创建，这两个函数会创建 node 实例并初始化相应的字段，其实现比较简单，代码不再贴出来，感兴趣的读者可以参考代码进行学习。

对于键值对节点，node 提供了 Read()和 Write()方法用来读写对应的值，需要注意的是，只能读写键值对节点，它们的具体实现如下：

```
func (n *node) Read() (string, *etcdErr.Error) {
  if n.IsDir() { // 如果当前节点是目录节点，则抛出异常
    return "", etcdErr.NewError(etcdErr.EcodeNotFile, "", n.store.CurrentIndex)
  }
  return n.Value, nil // 返回 Value
}
```

```
func (n *node) Write(value string, index uint64) *etcdErr.Error {
  // 如果当前节点是目录节点，则抛出异常(略)
  n.Value = value  // 更新值
  n.ModifiedIndex = index  // 记录更新时对应的 CurrentIndex 值
  return nil
}
```

当其他模块调用 v2 存储的 Storage 接口获取节点数据时，v2 存储并不会直接将相应的 node 实例暴露出去，而是会将 node 实例封装成 NodeExtern 实例之后再返回。结构体 NodeExtern 中除了包含 node 中的核心数据，还包含了一些扩展字段，这里先简单介绍一下 NodeExtern 中的核心字段含义。

- Key（string 类型）：对应 node 实例中的 Path 字段。为了实现排序功能，NodeExtern 实现了 sort 接口，在其 Less()方法实现中比较的就是 Key 字段。
- Value（*string 类型）：对应键值对节点中的 Value 字段。
- Dir（bool 类型）：对应节点是否为目录节点。
- Expiration（*time.Time 类型）：对应节点的过期时间。如果对应节点是“永久节点”，则该字段值为 nil。
- TTL（int64 类型）：对应节点的剩余存活时间，单位是秒。如果对应节点是“永久节点”，则该字段值为 0。
- Nodes（NodeExterns 类型）：子节点对应的 NodeExtern 实例。
- ModifiedIndex（uint64 类型）：对应节点的 ModifiedIndex 字段值。
- CreatedIndex（uint64 类型）：对应节点的 CreatedIndex 字段值。

在 node.Repr()方法中，会将当前节点和子节点（根据参数决定是否递归处理子节点以及子节点是否排序）转换成 NodeExtern 实例返回，其具体实现如下：

```
func (n *node) Repr(recursive, sorted bool, clock clockwork.Clock) *NodeExtern {
  if n.IsDir() { // 下面是针对目录节点的处理
    node := &NodeExtern{ // NodeExtern 是 node 的外部表示形式
      Key:          n.Path,
      Dir:          true,
      ModifiedIndex: n.ModifiedIndex,
      CreatedIndex:  n.CreatedIndex,
    }
```

```
    // 计算当前节点的过期时间和 TTL(即当前节点还能存活多长时间)
    node.Expiration, node.TTL = n.expirationAndTTL(clock)
    if !recursive { // recursive 参数表示是否递归获取子节点
      return node
    }
    children, _ := n.List() // 获取子节点
    node.Nodes = make(NodeExterns, len(children))
    i := 0
    for _, child := range children { // 遍历子节点
      if child.IsHidden() { // 忽略隐藏节点
        continue
      }
      // 调用子节点的 Repr()方法，创建对应的 NodeExtern 实例
      node.Nodes[i] = child.Repr(recursive, sorted, clock)
      i++
    }
    node.Nodes = node.Nodes[:i] // 这里简单进行压缩，释放隐藏节点导致的空间浪费
    if sorted { // 根据 sorted 参数，决定是否对子节点进行排序
      sort.Sort(node.Nodes)
    }
    return node
  }

  // 下面是针对 KV 节点的处理
  value := n.Value // 复制当前节点的 Value 值
  node := &NodeExtern{ // 创建当前节点对应的 NodeExtern 实例，并初始化相应字段
    Key:          n.Path,
    Value:        &value,
    ModifiedIndex: n.ModifiedIndex,
    CreatedIndex:  n.CreatedIndex,
  }
  node.Expiration, node.TTL = n.expirationAndTTL(clock) // 计算当前节点的过期时间和 TTL
  return node
}
```

node.expirationAndTTL()方法提供了计算当前节点存活时间的功能，其具体实现如下：

```
func (n *node) expirationAndTTL(clock clockwork.Clock) (*time.Time, int64) {
  if !n.IsPermanent() { // 检查当前节点是否为永久节点
    ttlN := n.ExpireTime.Sub(clock.Now()) // 节点过期时间减去当前时间，即为剩余的存活时间
    ttl := ttlN / time.Second
    if (ttlN % time.Second) > 0 { // 以秒为单位，不足一秒则按一秒算
      ttl++
    }
    t := n.ExpireTime.UTC()
    return &t, int64(ttl) // 返回当前节点的过期时间和剩余存活时间
  }
  return nil, 0 // 永久节点对应的返回值
}
```

node.List()方法主要负责获取当前节点（目录节点）的子节点，其具体实现如下：

```
func (n *node) List() ([]*node, *etcdErr.Error) {
  // 检测当前节点是否为目录节点，如果当前节点不是目录节点，则抛出异常(略)
  nodes := make([]*node, len(n.Children))
  i := 0
  for _, node := range n.Children { // 获取子节点
    nodes[i] = node
    i++
  }
  return nodes, nil
}
```

node 结构体不仅提供了 List()方法获取当前节点的全部子节点，还提供了 GetChild()方法获取指定的子节点，该方法直接从 Child 字段中获取相应的节点，实现比较简单，感兴趣的读者可以参考源码进行学习。

下面来分析 node.Add()和 Remove()方法，Add()方法的主要功能是在当前节点下添加子节点，具体实现如下：

```
func (n *node) Add(child *node) *etcdErr.Error {
  // 检测当前节点是否为目录节点(略)
  _, name := path.Split(child.Path) // 按照“/”进行切分，得到待添加子节点对应的 key
  // 检测当前节点的 Child 字段中是否已存在待添加子节点(略)
  n.Children[name] = child // 添加子节点
  return nil
}
```

Remove()方法负责将当前节点从其父节点中删除，其中会根据参数决定是否递归删除当前节点的子节点，其具体实现如下：

```
func (n *node) Remove(dir, recursive bool, callback func(path string)) *etcdErr.Error {
  if !n.IsDir() { // 当前节点为 KV 节点
    _, name := path.Split(n.Path) // 获取当前节点对应的 Key
    // 查找当前节点的父节点，并将当前节点从父节点的 Child 字段中删除
    if n.Parent != nil && n.Parent.Children[name] == n {
      delete(n.Parent.Children, name)
    }
    if callback != nil { // 回调 callback 函数
      callback(n.Path)
    }

    // 当前节点为非永久节点，则将其从 store.ttlKeyHeap 中删除，
    // store.ttlKeyHeap 中的节点是按照过期时间排序的，TTLKeyHeap 的具体实现后面详细介绍
    if !n.IsPermanent() {
      n.store.ttlKeyHeap.remove(n)
    }
    return nil
  }

  // 下面是针对目录节点的删除操作
  // 只有 dir 参数设置为 true，才能删除该目录节点
  if !dir {
    return etcdErr.NewError(etcdErr.EcodeNotFile, n.Path, n.store.CurrentIndex)
  }
  // 如果当前目录节点还存在子节点，且不能进行递归删除(recursive 参数设置为 false)，则不能删除该
  // 目录节点
  if len(n.Children) != 0 && !recursive {
    return etcdErr.NewError(etcdErr.EcodeDirNotEmpty, n.Path, n.store.CurrentIndex)
  }
  // 递归删除当前目录节点的全部子节点
  for _, child := range n.Children { // delete all children
    child.Remove(true, true, callback)
  }
  // 下面是删除的当前节点本身，该代码片段与前面 KV 节点的删除过程相同，所以这里省略了
```

```
    return nil
}
```

node 结构体提供的最后一个需要介绍的方法是 UpdateTTL()方法，该方法会更新指定节点的过期时间，另外还会更新该节点在 TTLKeyHeap 中的排序，具体实现如下：

```
func (n *node) UpdateTTL(expireTime time.Time) {
    if !n.IsPermanent() { // 对非永久节点的处理
        n.ExpireTime = expireTime // 更新当前节点的过期时间
        n.store.ttlKeyHeap.update(n) // 更新当前节点在 TTLKeyHeap 中排序
        return
    }
    // 下面是将永久节点变成非永久节点
    n.ExpireTime = expireTime
    n.store.ttlKeyHeap.push(n) // 将当前节点添加到 TTLKeyHeap 中
}
```

除了上述的普通的节点，v2 存储将以下画线开头的节点定义为“隐藏节点”。在后面介绍 storage.Get()时可以看到，它会将隐藏节点从查找的结果集中过滤掉，其中通过 node.IsHidden() 方法判断某个节点是否为“隐藏节点”，具体实现如下：

```
func (n *node) IsHidden() bool {
    _, name := path.Split(n.Path) // 按照最后一个“/”符号进行切分，获取节点所在的目录以及节
                                  // 点名称
    return name[0] == '_'   // 检查当前节点是否以下画线开头
}
```

6.1.2 Event

当 v2 存储收到调用请求之后，会进行相应的处理，在处理完成之后，会将处理结果及相关信息封装成 Event 实例返回。这里先来介绍一下 Event 中各个字段的含义。

- Action (string 类型)：该 Event 实例对应的操作，可选项有 Get、Create、Set、Update、Delete、CompareAndSwap、CompareAndDelete 和 Expire。
- Node (*NodeExtern 类型)：当前操作节点对应的 NodeExtern 实例。
- PrevNode (*NodeExtern 类型)：如果是更新操作，则记录该节点之前状态对应的 NodeExtern 实例。

- EtcdIndex（uint64 类型）：记录操作完成之后的 CurrentIndex 值。
- Refresh（bool 类型）：如果是 Set、Update、CompareAndSwap 三种涉及值更新的操作，则该字段都有可能被设置 true。当该字段被设置为 true 时，表示该 Event 实例对应的修改操作只进行刷新操作（例如，只修改了节点的过期时间），并没有改变节点的值，不会触发相关的 watcher。

Event 提供的方法都比较简单，这里就不再展开进行详细分析了，感兴趣的读者可以参考源码进行学习。

另外两个与 Event 相关的结构体分别是 eventQueue 和 EventHistory。eventQueue 是一个先进先出的、环形的 Event 队列，其中各个字段的含义如下。

- Events（[]*Event 类型）：底层真正存储 Event 实例的数组。
- Capacity（int 类型）：Events 字段的长度。
- Size（int 类型）：当前 Events 字段中存储的 Event 实例个数。
- Front（int 类型）：eventQueue 队列中第一个元素的下标值。
- Back（int 类型）：eventQueue 队列中最后一个元素的下标值。

还有一点需要注意的是，当 eventQueue 被填满之后，继续向其中添加 Event，则会导致最先添加的 Event 实例被覆盖，这一点在 eventQueue.insert()方法的代码中就可以看出，其具体实现如下：

```
func (eq *eventQueue) insert(e *Event) {
  eq.Events[eq.Back] = e // 添加 Event 实例
  eq.Back = (eq.Back + 1) % eq.Capacity // 后移 Back

  if eq.Size == eq.Capacity { // 当队列满了的时候，则后移 Front，抛弃 Front 指向的 Event 实例
    eq.Front = (eq.Front + 1) % eq.Capacity
  } else {
    eq.Size++ // 当队列没满的时候，增加 Size
  }
}
```

EventHistory 是对 eventQueue 的一层封装，其中字段的含义如下。

- Queue（eventQueue 类型）：用来存储 Event 实例的 eventQueue 实例。
- StartIndex（uint64 类型）：当前 EventHistory 实例中记录的第一个 Event 实例的 ModifiedIndex 字段值。

- LastIndex（uint64 类型）：当前 EventHistory 实例中记录的最后一个 Event 实例的 ModifiedIndex 字段值，从而可知，EventHistory 只记录了从 StartIndex～LastIndex 之间的 Event。
- rwl（sync.RWMutex 类型）：在添加或读取 Event 实例时，都需要通过该锁进行同步。

EventHistory.addEvent()方法提供了添加新 Event 实例的功能，其具体实现如下，

```
func (eh *EventHistory) addEvent(e *Event) *Event {
  eh.rwl.Lock() // 通过 rwl 锁进行同步
  defer eh.rwl.Unlock()
  eh.Queue.insert(e) // 添加 Event 实例
  eh.LastIndex = e.Index() // 更新 StartIndex 字段和 LastIndex 字段
  eh.StartIndex = eh.Queue.Events[eh.Queue.Front].Index()
  return e
}
```

EventHistory.scan()方法会从 index 参数指定的位置开始查找 EventHistory 中是否记录了参数 key 指定节点对应的 Event 实例，其具体实现如下：

```
func (eh *EventHistory) scan(key string, recursive bool, index uint64) (*Event,
*etcdErr.Error) {
    // 加锁和解锁的相关逻辑(略)
    // 检测 index 参数是否合法(即位于 StartIndex~LastIndex 之间)(略)
    // 首先将 index 参数转换成 eventQueue.Events 的下标
    offset := index - eh.StartIndex
    i := (eh.Queue.Front + int(offset)) % eh.Queue.Capacity

    for { // 遍历 index 之后的 Event 实例
      e := eh.Queue.Events[i]
      if !e.Refresh { // 过滤掉 Refresh 类型的修改操作
        ok := (e.Node.Key == key) // 根据 Key 匹配 Event 实例
        if recursive { // key 可能是个目录，根据 recursive 参数决定是否查找其子节点对应的 Event
          nkey := path.Clean(key)
          if nkey[len(nkey)-1] != '/' {
            nkey = nkey + "/"
          }
          ok = ok || strings.HasPrefix(e.Node.Key, nkey)
```

```
            }
            // 针对 Delete、Expire 等操作的处理，比较的是 Event.PrevNode
            if (e.Action == Delete || e.Action == Expire) &&
                    e.PrevNode != nil && e.PrevNode.Dir {
                ok = ok || strings.HasPrefix(key, e.PrevNode.Key)
            }
            if ok {
                return e, nil
            }
        }
        i = (i + 1) % eh.Queue.Capacity
        if i == eh.Queue.Back { // 整个 EventHistory 中不存在与参数 key 匹配的 Event 实例
            return nil, nil
        }
    }
}
```

6.1.3 watcher和watcherHub

watcher 是客户端用来监听 etcd 服务端数据变化的一种手段，在 ZooKeeper 等项目中也有类似的设计。在客户端对指定路径节点添加 watcher 之后，如果该节点数据发生变化，则 etcd 服务端会通知相应的客户端。

下面我们来看一下结构体 watcher 中各个字段的含义，如下所示。

- eventChan（chan *Event 类型）：当该 watcher 实例被修改操作触发时，会将对应的 Event 实例写入到该通道中，后续由网络层读取该通道，并通知客户端此次修改。在后面介绍 watcherHub 实现时会看到，在创建 watcher 实例时会将该通道的缓冲区设置成 100。
- hub（*watcherHub 类型）：在 watcherHub 中维护了该 watcher 实例与其监听的节点路径的映射关系，watcherHub 的具体实现在后面详细分析。
- sinceIndex（uint64 类型）：标识该 watcher 实例从哪个 CurrentIndex 值开始监听对应的节点。
- startIndex（uint64 类型）：记录创建该 watcher 实例时对应的 CurrentIndex 值。
- recursive（bool 类型）：标识当前 watcher 实例是否监听目标节点的子节点。
- removed（bool 类型）：标记当前 watcher 实例是否已经被删除。

- remove（func()类型）：用于删除当前 watcher 实例的回调函数，具体的调用时机在后面详细介绍。
- stream（bool 类型）：标识当前 watcher 是否为 stream 类型的，当前 stream watcher 被触发一次后，并不会被自动删除，而是继续保持监听，并返回一系列监听到的 Event。但是笔者从 GitHub 中的讨论来看，etcd v2 版本对 stream watcher 的支持并不是很好，如果有类似需求，笔者建议使用 etcd v3 版本实现。

watcher 最核心的方法是 notify()方法，该方法负责将触发该 watcher 的 Event 实例写入 eventChan 通道中，在如下三种场景下当前 watcher 会被触发：

- 当修改操作发生在当前 watcher 实例监听的节点上时，会触发该 watcher 实例。
- 当前 watcher 实例不仅监听当前节点的变化，同时也监听其子节点的变化，当修改操作发生在子节点上时，也会触发该 watcher 实例。
- 当删除某个目录节点时，需要通知在其子节点上监听的全部 watcher 实例。

watcher.notify()方法的具体实现如下：

```
func (w *watcher) notify(e *Event, originalPath bool, deleted bool) bool {
  if (w.recursive   // 检查当前 watcher 是否监听对应节点的子节点
      || originalPath    // 检查发生变化的是否正好是当前 watcher 监听节点
      || deleted   // 检查此次修改操作是否为删除操作
     )&& e.Index() >= w.sinceIndex { // 检查操作发生的时机是否为该 watcher 监听的范围
     select {
     case w.eventChan <- e: // 将修改操作对应的 Event 写入 eventChan 通道等待处理
     default:
        w.remove() // 如果当前 watcher.eventChan 通道被填充满了，则会将该通道关闭，这可能导致
                   // 事件丢失
     }
     return true
  }
  return false
}
```

在 watcherHub 创建 watcher 实例时，会将 watcher.remove 字段初始化为如下函数：

```
w.remove = func() {
  if w.removed { // 检测当前 watcher 实例是否已经被删除
     return
  }
```

```
    w.removed = true // 标记当前 watcher 已被删除
    l.Remove(elem) // 将当前 watcher 从 watcherHub 中删除
    atomic.AddInt64(&wh.count, -1) // 更新 watcherHub 中保存的 watcher 个数
    if l.Len() == 0 { // 如果没有任何监听某节点，则将该节点对应的 watcher 列表删除
      delete(wh.watchers, key)
    }
}
```

接下来，我们来分析一下 watcherHub 的大致实现，下面先通过图 6-2 了解 watcherHub 的结构。

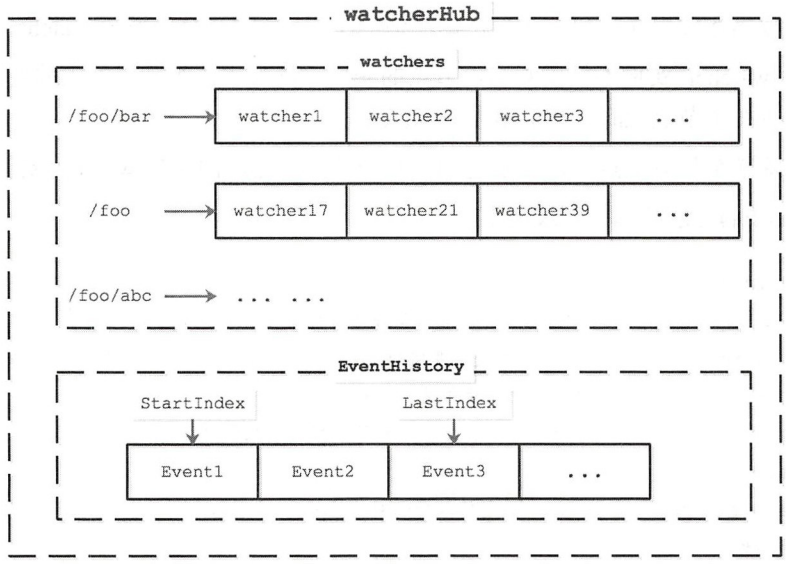


图 6-2

结构体 watcherHub 中各个字段的含义如下所示。

- count（int64 类型）：当前 watcherHub 实例中保存的 watcher 实例个数。
- watchers（map[string]*list.List 类型）：维护节点与监听该节点的 watcher 实例的对应关系，其中 key 是 node.Path 字段值，value 则是监听该节点的 watcher 实例列表。
- EventHistory（*EventHistory 类型）：保存最近发生的修改操作对应的 Event 实例，在前面的介绍中也提到过，当 EventHistory 的容量达到上限之后，继续向其中添加 Event 实例，则会自动删除最早添加的 Event 实例。

当客户端请求 etcd 服务端为某个指定的节点添加 watcher 时，是通过调用 watcherHub.watch() 方法完成的。在添加 watcher 之前，会先查找 EventHistory 确定 sinceIndex～CurrentIndex 之间是否发生了触发待添加 watcher 的操作。watcherHub.watch()方法的具体实现如下所示。

```
func (wh *watcherHub) watch(key string, recursive, stream bool, index,
        storeIndex uint64) (Watcher, *etcdErr.Error) {
  w := &watcher{
    eventChan:  make(chan *Event, 100), // use a buffered channel
    recursive:  recursive, // 根据 recursive 参数设置是否监听指定节点的子节点
    stream:     stream,
    sinceIndex: index, // 根据 index 参数设置从哪个事件开始监听
    startIndex: storeIndex, // 创建该 watcher 实例时对应的 CurrentIndex 值
    hub:        wh, // 关联的 watcherHub 实例
  }

  // 相关的加锁和解锁操作(略)
  // 在 EventHistory 中，从 index 开始查找对应的 Event，
  event, err := wh.EventHistory.scan(key, recursive, index)
  // 如果在 EventHistory 中存在合适的 Event，则表示从 sinceIndex 到 CurrentIndex 为止，
  // 发生了触发该 watcher 的操作
  if event != nil {
    ne := event.Clone()
    ne.EtcdIndex = storeIndex
    // 将查找到的 Event 写入 eventChan 通道中等待处理
    w.eventChan <- ne
    return w, nil // 直接返回 watcher
  }
  // 如果未查找到合适的 Event，则表示从 sinceIndex 到 CurrentIndex 为止，并没有发生过
  // 触发该 watcher 的操作
  l, ok := wh.watchers[key] // 查找节点对应的 watcher 列表
  var elem *list.Element

  if ok { // 将当前 watcher 添加到列表尾部
    elem = l.PushBack(w)
  } else { // 第一次为节点添加 watcher 时，会先创建对应的 watcher 列表
    l = list.New()
    elem = l.PushBack(w)
    wh.watchers[key] = l
  }

  // 初始化当前 watcher.remove 字段，前面已经介绍过了，这里不再赘述(略)
```

```
    atomic.AddInt64(&wh.count, 1) // 更新当前 watcherHub 中记录的 watcher 数量
    return w, nil
}
```

介绍完添加 watcher 的逻辑之后，我们来分析发生修改操作时对应的处理逻辑。storage 会通过 watcherHub.notify()方法触发监听对应节点的 watcher，其具体实现如下：

```
func (wh *watcherHub) notify(e *Event) {
    // 首先将修改操作对应的 Event 实例保存到 EventHistory 中
    e = wh.EventHistory.addEvent(e) // add event into the eventHistory
    segments := strings.Split(e.Node.Key, "/") // 按照“/”切分发生修改的节点的路径
    currPath := "/"

    // 遍历切分结果，并调用 notifyWatchers()方法，通知当前节点和所有父节点上的 watcher，
    // 这里举个例子，如果“/foo/bar”节点被修改了，那么不仅监听“/foo/bar”节点的 watcher 会被
    // 触发，监听“/foo”节点的 watcher 也可能被触发
    for _, segment := range segments {
        currPath = path.Join(currPath, segment)
        wh.notifyWatchers(e, currPath, false)
    }
}
```

接下来继续分析 watcher.notifyWatchers()方法，该方法会查找节点对应的 watcher 列表，并逐一触发列表中的 watcher 实例，当 watcher 实例（stream watcher 除外）被成功触发之后，则会将其清除，notifyWatchers()方法的具体实现如下：

```
func (wh *watcherHub) notifyWatchers(e *Event, nodePath string, deleted bool) {
    wh.mutex.Lock()
    defer wh.mutex.Unlock()

    // 省略加锁和解锁的相关逻辑
    l, ok := wh.watchers[nodePath] // 获取监听指定节点的 watcher 列表
    if ok {
        curr := l.Front()
        for curr != nil { // 遍历该 watcher 列表
            next := curr.Next()
            w, _ := curr.Value.(*watcher) // 获取 watcher 实例
            originalPath := (e.Node.Key == nodePath)
```

```
            // 调用 watcher.notify()方法，触发相应的 watcher
            if (originalPath || !isHidden(nodePath, e.Node.Key))
                    && w.notify(e, originalPath, deleted) {
               if !w.stream { // 检测是否为 stream watcher，如果不是，则需要将该 watcher 清除
                  w.removed = true
                  l.Remove(curr) // 从当前 watcher 列表中删除
                  atomic.AddInt64(&wh.count, -1) // 更新 count 字段
                  reportWatcherRemoved()
               }
            }
            curr = next // 继续处理 watcher 列表中的下一个 watcher 实例
         }

         if l.Len() == 0 { // 如果列表中的 watcher 实例全部被清空，则将该列表从 watchHub 中清除
            delete(wh.watchers, nodePath)
         }
      }
   }
```

6.1.4 Store

本小节主要介绍 Store 接口及其相关实现，Store 接口是 v2 存储对外暴露的 API 接口，其中定义了 v2 存储的主要行为。在本小节中分析 Store 接口实现时，会详细介绍 Store 接口中每个方法的含义，这里不再单独罗列这些方法的定义。

结构体 store 实现了这里的 Store 接口，其中各字段的含义如下所示。

- Root (*node 类型)：在前文中提到，v2 存储是纯内存实现，它以树形结构将全部数据维护在内存中，树中的每个节点都是前面介绍的 node 实例。该字段记录了此树型结构的根节点。
- WatcherHub (*watcherHub 类型)：watcherHub 的主要功能是管理客户端添加的 watcher 监听和 Event 实例，其具体实现在上一小节中已经详细介绍过了，这里不再赘述。
- CurrentIndex (uint64 类型)：该字段是修改操作的唯一标识，每出现一次修改操作，该字段就会自增一次。
- ttlKeyHeap (*ttlKeyHeap 类型)：ttlKeyHeap 的主要功能是将全部节点按照过期时间进行排序，形成一个最小堆。ttlKeyHeap 中有两个字段，这里简单介绍一下。

  - array（[]*node 类型）：在 ttlKeyHeap 底层就是通过该数组维护其最小堆的结构，其中存储了全部的节点。
  - keyMap（map[*node]int 类型）：为了快速更新排序或是删除节点，在该 map 中维护了 node 节点到其下标索引的映射关系。

  ttlKeyHeap 提供的其他方法都比较简单，这里不再展开详细介绍，感兴趣的读者可以参考代码。

- worldLock（sync.RWMutex 类型）：在 store 进行任何操作之前，都需要获取该锁进行同步。
- readonlySet（types.Set 类型）：记录了哪些节点是只读节点，这些节点都无法被修改。

在下面的几个小节中，我们将重点介绍 store 结构体提供的方法。

1. Get 方法

我们首先来分析 store.Get()方法，该方法的主要功能就是在树形结构中查找指定路径对应的 node 节点，其中会根据 recursive 参数和 sorted 参数决定是否加载子节点，以及是否对子节点进行排序，Get()方法的具体实现如下：

```
func (s *store) Get(nodePath string, recursive, sorted bool) (*Event, error) {
  var err *etcdErr.Error
  s.worldLock.RLock() // 加锁
  defer s.worldLock.RUnlock() // 方法结束时解锁

  // 根据 nodePath 获取对应 node 节点，省略异常处理的相关代码
  n, err := s.internalGet(nodePath)

  // 创建 Event 实例
  e := newEvent(Get, nodePath, n.ModifiedIndex, n.CreatedIndex)
  e.EtcdIndex = s.CurrentIndex
  // 如果待查找节点是目录节点，则获取子节点；如果是 KV 节点，则加载其 Value 值
  e.Node.loadInternalNode(n, recursive, sorted, s.clock)
  return e, nil
}
```

store.Get()方法查找指定 node 节点的功能是通过 internalGet()方法实现的。在 internalGet()方法中，会根据给定的路径从 Root 节点逐层查找，直至查找到目标 node 节点，其具体实现如下：

```
func (s *store) internalGet(nodePath string) (*node, *etcdErr.Error) {
  nodePath = path.Clean(path.Join("/", nodePath)) // 整理路径格式
  // 定义 walkFunc()方法，其功能是在 parent 节点下查找指定子节点，若查找失败，则返回异常
  walkFunc := func(parent *node, name string) (*node, *etcdErr.Error) {
    if !parent.IsDir() { // 检测 parent 节点是否为目录节点
      err := etcdErr.NewError(etcdErr.EcodeNotDir, parent.Path, s.CurrentIndex)
      return nil, err
    }
    child, ok := parent.Children[name] // 查找指定的子节点并返回，查找失败，则返回异常
    if ok {
      return child, nil
    }
    return nil, etcdErr.NewError(etcdErr.EcodeKeyNotFound,
        path.Join(parent.Path, name), s.CurrentIndex)
  }

  f, err := s.walk(nodePath, walkFunc) // 真正执行 walkFunc()方法，逐层查找节点的地方
  return f, nil
}
```

我们继续深入分析 walk()方法的具体实现，如下所示。

```
func (s *store) walk(nodePath string, walkFunc func(prev *node, component string)
      (*node, *etcdErr.Error)) (*node, *etcdErr.Error) {
  components := strings.Split(nodePath, "/")
  curr := s.Root // 从 Root 节点开始查找
  for i := 1; i < len(components); i++ {
    if len(components[i]) == 0 { // 忽略空路径
      return curr, nil
    }
    // 通过前面介绍的 walkFunc()函数查找 curr 节点的指定子节点
    curr, err = walkFunc(curr, components[i])
  }
  return curr, nil
}
```

2. Create 方法

store.Create()方法对外提供了创建 node 节点的功能，在其创建指定节点的过程中，还会同时创建中间不存在的目录节点（这些目录节点会被设置成永久的）。store.Create()方法的具体实现如下：

```
func (s *store) Create(nodePath string, dir bool, value string,
      unique bool, expireOpts TTLOptionSet) (*Event, error) {
  var err *etcdErr.Error
  // 加锁和解锁的相关代码(略)
  // 创建目标节点，同时会创建中间涉及的目录节点
  e, err := s.internalCreate(nodePath, dir, value, unique,
     false, expireOpts.ExpireTime, Create)
  if err != nil {
    return nil, err
  }
  e.EtcdIndex = s.CurrentIndex // 设置 Event.EtcdIndex
  s.WatcherHub.notify(e) // 将 Event 添加到 EventHistory 中，同时触发相关的 watcher
  return e, nil
}
```

store.internalCreate()方法是创建节点的核心，先来了解该方法中各个参数的含义，如下所示。

- nodePath：待创建节点的完整路径。
- dir：此次创建的节点是否为目录节点。
- value：如果此次创建的节点为键值对节点，则 value 为其值。
- unique：是否要创建一个唯一节点。
- replace：待创建的节点已存在，是否要对其进行替换。注意，这里只能替换已存在的键值对节点，不能替换已存在的目录节点。
- expireTime：待创建节点的过期时间。

下面我们深入 internalCreate()方法的具体实现进行分析，如下所示。

```
func (s *store) internalCreate(nodePath string, dir bool, value string,
 unique, replace bool, expireTime time.Time, action string) (*Event, *etcdErr.Error) {
  currIndex, nextIndex := s.CurrentIndex, s.CurrentIndex+1
  if unique {
    // 根据 unique 参数决定是否创建唯一节点，其方式在指定路径下，以 nextIndex 为名称创建一个节点
```

```
    nodePath += "/" + fmt.Sprintf("%020s", strconv.FormatUint(nextIndex, 10))
}
nodePath = path.Clean(path.Join("/", nodePath)) // 整理路径
// 检测待创建节点是否在只读路径下，如果是，则抛出异常(略)

if expireTime.Before(minExpireTime) { // 设置节点的过期时间
    expireTime = Permanent
}

dirName, nodeName := path.Split(nodePath) // 切分路径得到父节点路径及待创建节点的名称
// 遍历整个路径，如果路径中有目录节点不存在，则创建该目录节点，省略异常处理的相关代码
// 注意，这里的返回值是待创建节点的父节点，例如，待创建节点是“/foo/bar/tom”，则此处返回值
// “/foo/bar”节点，checkDir()方法的具体实现在下面会进行介绍
d, err := s.walk(dirName, s.checkDir)

// 创建此次操作对应的 Event 实例
e := newEvent(action, nodePath, nextIndex, nextIndex)
eNode := e.Node

n, _ := d.GetChild(nodeName) // 查找待创建节点
if n != nil { // 如果待创建节点已经存在，则根据 replace 参数决定是否替换已存在的节点
    if replace {
        if n.IsDir() {
            return nil, etcdErr.NewError(etcdErr.EcodeNotFile, nodePath, currIndex)
        }
        e.PrevNode = n.Repr(false, false, s.clock) // 将已存在节点的信息记录到
                                                   // Event.PrevNode 中
        n.Remove(false, false, nil) // 删除已存在的节点
    } else { // 如果不替换已存在的节点，则直接返回异常
        return nil, etcdErr.NewError(etcdErr.EcodeNodeExist, nodePath, currIndex)
    }
}

if !dir { // 根据 dir 参数决定创建 KV 节点还是目录节点
    valueCopy := value
    eNode.Value = &valueCopy
    n = newKV(s, nodePath, value, nextIndex, d, expireTime)
} else { // 创建目录节点
```

```
        eNode.Dir = true
        n = newDir(s, nodePath, nextIndex, d, expireTime)
    }
    d.Add(n) // 将创建好的节点添加到父节点中
    if !n.IsPermanent() { // 如果新建节点是非永久节点，则将其记录到 ttlKeyHeap 中
        s.ttlKeyHeap.push(n)
        // 更新 Event 中的过期时间和存活时间
        eNode.Expiration, eNode.TTL = n.expirationAndTTL(s.clock)
    }
    s.CurrentIndex = nextIndex // 递增 CurrentIndex
    return e, nil
}
```

store.walk()方法在前面已经分析过了，它会遍历指定路径上的每个目录节点，并调用传入的 walkFunc()函数，这里不再赘述。在这里传入的 walkFunc()函数是 store.checkDir()，该方法负责查找指定节点下的指定子节点，如果子节点不存在，则创建对应的目录节点。store.checkDir()方法的具体实现如下：

```
func (s *store) checkDir(parent *node, dirName string) (*node, *etcdErr.Error) {
    node, ok := parent.Children[dirName] // 在父节点中查找指定的子节点

    if ok {
        if node.IsDir() { // 如果自己子节点是目录节点，则返回子节点
            return node, nil
        }
        // 如果子节点不是目录节点，则直接返回异常，因为无法在此节点下新建其他节点
        return nil, etcdErr.NewError(etcdErr.EcodeNotDir, node.Path, s.CurrentIndex)
    }
    // 如果没有查找到对应节点，则创建对应的目录节点，并添加到父节点中
    n := newDir(s, path.Join(parent.Path, dirName), s.CurrentIndex+1, parent, Permanent)
    parent.Children[dirName] = n
    return n, nil // 返回的依然是子节点
}
```

3. Set 方法

store.Set()方法对外提供了创建（或更新）node 节点的功能，如果指定的节点已存在，则会将其替换。store.Set()方法首先会调用 internalGet()方法查找指定节点，然后根据 expireOpts 参数决定此次修改操作的类型，之后调用 internalCreate()方法替换现有节点，最后将此次操作的相关

信息封装成 Event 实例保存，并决定是否触发相关 watcher 实例。store.Set()方法具体实现如下：

```
func (s *store) Set(nodePath string, dir bool, value string,
    expireOpts TTLOptionSet) (*Event, error) {
  // 加锁和解锁的相关代码(略)
  // 通过 internalGet()方法获取指定节点，其实现在前面已经介绍过了，这里不再介绍
  n, getErr := s.internalGet(nodePath)
  // 处理 internalGet()方法在操作过程中遇到的异常，注意，这里会忽略 EcodeKeyNotFound 异常(略)

  if expireOpts.Refresh { // 根据此次操作类型，决定最终的节点值
    // 异常处理，不会忽略 EcodeKeyNotFound 异常，因为节点都没找到，所以无法获取其原值(略)
    value = n.Value // 如果是 Refresh 类型的修改操作，则不会改变节点的值
  }

  // 通过 internalCreate()方法创建指定的节点，其实现在前面已经介绍过了，这里不再介绍
  // 这里需要注意其 replace 参数，该值被设置为 true
  e, err := s.internalCreate(nodePath, dir, value, false, true,
      expireOpts.ExpireTime, Set)
  e.EtcdIndex = s.CurrentIndex

  // 设置此次修改操作对应 Event 的 PrevNode 字段，将其指向修改之前的节点
  if getErr == nil {
    prev := newEvent(Get, nodePath, n.ModifiedIndex, n.CreatedIndex)
    prev.Node.loadInternalNode(n, false, false, s.clock)
    e.PrevNode = prev.Node
  }

  if !expireOpts.Refresh {
    s.WatcherHub.notify(e)// 如果此次修改操作不是 Refresh 类型的，则触发相关的 watcher
  } else {
    // 此次修改操作为 Refresh 类型，则不会触发任何 watcher，而只是将其添加到 EventHistory 中保存
    e.SetRefresh()
    s.WatcherHub.add(e)
  }
  return e, nil
}
```

4. Update 方法

store.Update()方法对外提供了更新 node 节点的功能。如果待更新节点是一个键值对节点，那么我们可以同时更新其 Value 值和过期时间。如果待更新的节点时一个目录节点，那么我们只能更新其过期时间。store.Update()方法的具体实现如下：

```
func (s *store) Update(nodePath string, newValue string,
      expireOpts TTLOptionSet) (*Event, error) {
  // 省略加锁解锁的相关代码
  nodePath = path.Clean(path.Join("/", nodePath)) // 整理路径
  // 检测待更新节点是否为只读的(略)
  currIndex, nextIndex := s.CurrentIndex, s.CurrentIndex+1
  // 调用 internalGet()方法，查找待更新节点，省略异常处理的相关代码
  n, err := s.internalGet(nodePath)

  // 检测待更新节点是否为目录节点，如果是，则不能更新其 Value
  if n.IsDir() && len(newValue) != 0 {
    return nil, etcdErr.NewError(etcdErr.EcodeNotFile, nodePath, currIndex)
  }

  // 如果此次操作是 Refresh 类型的，则其 Value 值不会更新，只会更新其过期时间
  if expireOpts.Refresh {
    newValue = n.Value
  }
  // 创建此次操作对应的 Event 实例，并更新相关字段
  e := newEvent(Update, nodePath, nextIndex, n.CreatedIndex)
  e.EtcdIndex = nextIndex
  e.PrevNode = n.Repr(false, false, s.clock)
  eNode := e.Node
  // 更新节点的 Value 值和 ModifiedIndex 字段
  n.Write(newValue, nextIndex)
  // 更新节点的过期时间和对应 Event 实例中的 TTL
  n.UpdateTTL(expireOpts.ExpireTime)
  eNode.Expiration, eNode.TTL = n.expirationAndTTL(s.clock)

  if !expireOpts.Refresh { // 此次修改操作不是 Refresh 类型，则触发相关 watcher
    s.WatcherHub.notify(e)
  } else { // 此次修改操作是 Refresh 类型，则不会触发相关 watcher，只是将其添加到 EventHistory 中
    e.SetRefresh()
```

```
        s.WatcherHub.add(e)
    }
    s.CurrentIndex = nextIndex // 更新 store.CurrentIndex 字段
    return e, nil
}
```

5. Delete 方法

store.Delete()方法对外提供了删除指定 node 节点的功能。如果待删除节点是目录节点，则需要将 recursive 参数设置为 true，才能递归删除其子节点，并最终删除该目录节点。store.Delete()方法的具体实现如下：

```
func (s *store) Delete(nodePath string, dir, recursive bool) (*Event, error) {
    // 加锁和解锁的相关代码(略)
    // 检测待删除节点是否为只读的(略)
    if recursive {
        dir = true
    }
    // 调用 internalGet()方法获取待删除的节点
    n, err := s.internalGet(nodePath)

    nextIndex := s.CurrentIndex + 1
    // 创建此次删除操作对应的 Event 实例
    e := newEvent(Delete, nodePath, nextIndex, n.CreatedIndex)
    e.EtcdIndex = nextIndex
    e.PrevNode = n.Repr(false, false, s.clock)
    eNode := e.Node

    if n.IsDir() {
        eNode.Dir = true
    }

    callback := func(path string) { // 回调函数
        s.WatcherHub.notifyWatchers(e, path, true)
    }
    // 删除指定节点，如果是目录节点，则会根据 recursive 参数决定是否删除子节点，
    // 并在删除过程中调用回调函数，其中会触发相关 watcher
    err = n.Remove(dir, recursive, callback)
    if err != nil {
```

```
        return nil, err
    }
    s.CurrentIndex++ // 递增 store.CurrentIndex 值
    s.WatcherHub.notify(e) // 触发相关 watcher
    return e, nil
}
```

6. CompareAndSwap 方法

store.CompareAndSwap()方法对外提供了 CAS 操作，其主要流程是先查找待处理节点，然后比较节点的当前值与传入的 prevValue，同时会比较当前节点的 ModifiedIndex 与传入的 prevIndex，如果相等则表示当前节点没有被修改过，此时就会对节点的值进行修改；如果不相等则表示当前节点已经被别人修改过，此时不应该对其进行修改。store.CompareAndSwap()方法的具体实现如下：

```
// 这里简单说明一下 CompareAndSwap()方法中的 prevValue 参数，调用者认为目标节点当前值应该是
// prevValue，如果当前节点被修改过，则不再是 prevValue，则调用者不能对其进行修改；prevIndex 同理
func (s *store) CompareAndSwap(nodePath string, prevValue string, prevIndex uint64,
        value string, expireOpts TTLOptionSet) (*Event, error) {
    // 加锁和解锁的相关代码(略)
    // 检测待删除节点是否为只读的(略)
    // 调用 internalGet()方法获取待处理的节点
    n, err := s.internalGet(nodePath)
    if n.IsDir() { // 检测目标节点是否为目录节点，CAS 操作只能修改 KV 节点
        err = etcdErr.NewError(etcdErr.EcodeNotFile, nodePath, s.CurrentIndex)
        return nil, err
    }

    // 比较目标节点的值和 prevValue，同时也会比较当前节点的 ModifiedIndex 和 prevIndex
    if ok, which := n.Compare(prevValue, prevIndex); !ok {
        // 如果任意一个字段不相等，则无法更新，返回异常(略)
    }
    // 若上述两个比较完全相等，则继续后面的修改操作
    if expireOpts.Refresh { // 如果此次更新是 Refresh 类型，则节点的值不变
        value = n.Value
    }

    s.CurrentIndex++ // 递增 store.CurrentIndex
    // 创建此次 CAS 操作对应的 Event 实例，并更新其字段
```

```
    e := newEvent(CompareAndSwap, nodePath, s.CurrentIndex, n.CreatedIndex)
    e.EtcdIndex = s.CurrentIndex
    e.PrevNode = n.Repr(false, false, s.clock)
    eNode := e.Node

    n.Write(value, s.CurrentIndex) // 更新节点值
    n.UpdateTTL(expireOpts.ExpireTime) // 更新节点的过期时间

    valueCopy := value
    eNode.Value = &valueCopy
    eNode.Expiration, eNode.TTL = n.expirationAndTTL(s.clock)

    // 根据更新操作是否为 Refresh 类型，决定是否触发相应的 watcher
    if !expireOpts.Refresh {
      s.WatcherHub.notify(e)
    } else {
      e.SetRefresh()
      s.WatcherHub.add(e)
    }
    return e, nil
}
```

store.CompareAndDelete()方法与 CompareAndSwap()方法的原理及具体实现大致相同，这里不再展开介绍，感兴趣的读者可以参考源码进行学习。

7. DeleteExpiredKeys 方法

store.DeleteExpiredKeys()方法的主要功能是删除过期节点。前面介绍过 ttlKeyHeap 的主要功能和实现，DeleteExpiredKeys()方法通过其维护的最小堆来按序清理过期节点。DeleteExpiredKeys()方法的具体实现如下：

```
func (s *store) DeleteExpiredKeys(cutoff time.Time) {
    // 加锁和解锁的相关代码(略)
    for {
      node := s.ttlKeyHeap.top() // 获取过期时间最近的节点
      if node == nil || node.ExpireTime.After(cutoff) {
        break // 整个存储中都没有过期节点，则直接结束
      }
```

```
            s.CurrentIndex++ // 递增 store.CurrentIndex
            // 创建此次操作对应的 Event 实例
            e := newEvent(Expire, node.Path, s.CurrentIndex, node.CreatedIndex)
            e.EtcdIndex = s.CurrentIndex
            e.PrevNode = node.Repr(false, false, s.clock)
            if node.IsDir() { // 检测过期节点是否为目录节点
                e.Node.Dir = true
            }

            callback := func(path string) { // 回调函数，负责触发相关的 watcher
                s.WatcherHub.notifyWatchers(e, path, true)
            }

            s.ttlKeyHeap.pop() // 将过期节点从 ttlKeyHeap 中删除
            // 将过期节点从 store 中删除(如果是目录节点，则递归删除其子节点)
            node.Remove(true, true, callback)
            s.WatcherHub.notify(e) // 触发相关 watcher
        }
    }
```

在 store 结构体中还提供了 Save()和 Recovery()两个方法，前者负责将整个 store 实例序列化成 JSON 格式，后者负责将 JSON 格式的数据反序列化成 store 实例。这两个方法的实现比较简单，这里不再赘述，感兴趣的读者可以参考源码进行学习。

至此，etcd v2 版本的后端存储的具体实现就介绍完了。在 etcd 3.2 版本中，依然支持使用 v2 版本的后端存储保存数据。v2 版本后端存储存存在的缺陷在前面也详细介绍过了，当其无法满足需求时，推荐使用 etcd v3 版本的后端存储。在下一节将对 etcd v3 版本的后端存储进行详细的介绍。

6.2 etcd v3 版本存储

通过上一节对 etcd v2 版本后端存储的介绍我们知道，其实现是基于内存的，也没有提供事务相关的功能。另外，它最大的问题就是 EventHistroy 表的长度限制，以及触发 watcher 时的缓冲区大小限制，这两点都可能会造成相关的事件丢失。这些问题与其基于内存的实现方式有着莫大的关系，所以完全依赖其 watcher 机制来实现数据同步存在很大的风险。

6.2.1 backend

v3 版本的后端存储在一定程度上解决了上述问题，v3 版本存储的具体实现在 mvcc 包中。在 v3 版本存储中，将 watcher 监听和持久化存储分开进行实现，下面先来分析 store 的相关实现。store 分为两部分：

- 第一部分是 backend store，在 etcd v3 的设计中，backend store 可以使用多种不同的存储，默认使用的是 BoltDB。BoltDB 是一个单机的支持事务的键值对存储，etcd v3 存储中提供的事务就是基于 BoltDB 的事务实现的。
- 第二部分是内存索引，核心实现是 keyIndex。keyIndex 是基于 Google 开源的一个 btree 实现的。

1. BoltDB 简介

BoltDB 是根据 LMDB 项目开发的一个的 Go 语言版的 Key/Value 存储，它的目标是为应用提供一个简单、高效、可靠的嵌入式数据库存储，而不是一个完整的数据库服务。按照 BoltDB 官方文档的介绍，BoltDB 之所以具有较高的性能，与其底层实现息息相关：

- 为了让数据读取更加高效，BoltDB 使用操作系统提供的 mmap 技术，避免 I/O 操作。

  mmap 是一种内存映射磁盘文件的技术，它会将一个文件映射到应用程序的地址空间中，实现文件的磁盘地址与进程中的一段虚拟地址一一映射。这样，应用程序无须再调用 read、write 等系统调用进行文件的读写（先将文件内容复制到内核空间，然后将数据从内核空间复制到用户空间），而是通过指针直接读写该段内存即可，底层的操作系统会自动将脏页写回到对应的文件中。mmap 技术减少了文件内容的复制次数，从而提高了文件读写的效率。这里不再深入介绍 mmap 的实现原理，感兴趣的读者可以参考相关文章进行学习。
- 为了提高读操作的并发，使用写时复制技术（Copy On Write）。
- 为了提高随机读操作的性能，内部使用 B+树实现索引。
- BoltDB 使用 Go 语言开发，而且曾经被 InfluxDB 用做底层存储。

通过与其他同类产品的比较，可以帮助我们快速了解一个新产品的目标和定位。这里就将 BoltDB 与 LevelDB 进行简单的比较，LevelDB 及其衍生产品（例如，RocksDB、HyperLevelDB 等）都与 BoltDB 类似，它们都是绑定到应用程序中的数据库，但是它们底层依赖的数据结构与 BoltDB 有所不同。前者主要依赖 LSM 树实现，LSM 树通过使用日志和多层、排序的文件 SSTables 来优化随机写入，而 BoltDB 在内部使用 B+树，且它只有一个文件。如果在实践中随机写吞吐量较高，那么使用 LevelDB 是一个不错的选择。但是如果实践场景是读多写少，或者要进行大量的范围扫描操作，那么 BoltDB 可能是一个不错的选择。另一点需要注意的是，

LevelDB 是没有事务的，虽然它可以批量写入键值对，但它不会安全地进行比较和交换操作，BoltDB 则支持完全序列化的 ACID 事务。LevelDB 是 C++写的，但是也有些 Go 语言的实现方式，如 syndtr/goleveldb、leveldb-go。

在本小节介绍 BoltDB 的使用时，会涉及 BoltDB 中的几个数据结构，分别是 DB、Bucket、Tx、Cursor、node、inode 和 page。其中，DB 表示数据库中所有 Bucket 集合，Bucket 表示数据库中的键值对集合，好比 DB 是 MySQL，Bucket 是 MySQL 中的一张表，键值对则对应 MySQL 表中的一行数据。Cursor 表示一个迭代器，该迭代器可以按顺序遍历 Bucket 中的所有键值对，这与关系型数据库中的游标有类似之处。Tx 表示数据库中的一个只读或读写事务，只读事务可以用来创建 Cursor 读取数据。而读写事务除了读取数据，还可以创建、删除 Bucket 和键值对。node 表示一个在内存中的反序列化的 page，inode 是一个 node 中的内部节点，可以用来指向 page 中的元素或者还未被被添加到 page 中的元素。通常说的键值对就对应一个 inode。

在 BoltDB 中，无论是添加、删除还是读取，都需要先打开数据库文件，并获取一个事务。在只读事务中，不允许进行任更改，只能查找 Bucket 和键值对。在读写事务中则是可以进行数据修改的。在读写事务结束时，如果没有异常则提交事务，对应的数据将被更改，如果出现异常，则事务会回滚，该事务中所有的更改都会被丢弃。不管是读取、删除还是写入，都需要先通过 Cursor 来找到对应 Key 的位置，比如读取操作，需要先比较对应位置的键和目标键是否相等，如果相等则查找结束并且返回。

在 BoltDB 中维护了 B+树索引，B+树可以加速读操作。在提交事务时，如果有删除操作，则检查是否需要重新平衡该 B+树索引，如果有数据更改，则写入 Dirty Page，写入时还需要检测是否有 node 需要拆分，其中着重说明的是，有数据更新并且数据长度不一致，很可能导致数据所在 Page 的改变（默认情况下，BoltDB 的 Page 在使用率到达 50%时就会切页，大数据块也可能占据连续的几页，让使用率达到 100%）。

从源码的角度来看，BoltDB 的代码写得非常好，尤其是其中 B+树索引的相关代码，非常值得一读，可以让读者对 B+树的使用及数据存储有更深刻的认识，也会对数据库系统有更进一步的认识。

接下来将介绍 BoltDB 的 API 及简单使用。首先，我们通过 BoltDB 提供的 Open()方法可以创建 BoltDB 数据库，如下所示。

```
db, err := bolt.Open("my.db", 0600, nil)
if err != nil {
   log.Fatal(err)
}
defer db.Close()
```

其中第一个参数为数据库文件路径，如果数据库文件不存在则创建对应文件；第二个是操作数据库文件的权限；第三个参数是可选的，可以设置操作此数据库文件是否为只读模式、打开数据库文件的超时时间等参数。

需要特别注意的是，BoltDB.Open()方法会获取上述数据文件的文件锁，所以打开一个已经被打开的 BoltDB 文件会导致进程被挂起，直到另一个进程关闭释放对应的数据库文件，因此不能多进程同时打开同一数据库。另外，为了防止进程长时间的等待，在 Open()方法的第三个参数中设置超时时间，例如，我们可以将打开数据库文件的超时时间设置成一秒，具体如下所示。

```
db, err := bolt.Open("my.db", 0600, &bolt.Options{Timeout: 1 * time.Second})
```

在打开数据库之后，我们可以获取一个读写事务，然后进行一系列读写操作，并在相应的操作完成之后提交事务，这样，事务中的修改就可以持久化到数据库文件。需要特别注意的是，BoltDB 一次只允许一个读写事务，如果同时启动两个读写事务，则第二个读写事务会被阻塞，直至前者关闭为止。

要启动一个读写事务，可以通过 Update()方法实现，具体示例如下：

```
err := db.Update(func(tx *bolt.Tx) error {
    // 读写事务中的操作
    return nil
})
```

在这个闭包中包含了读写事务中的全部数据库操作，我们可以在结束时返回 nil 来提交该读写事务，也可以通过返回一个错误来回滚该读写事务。

我们可以使用 View()方法启动一个只读事务，具体示例如下：

```
err := db.View(func(tx *bolt.Tx) error {
    // 只读事务中的操作
    return nil
})
```

在这个闭包中，我们可以得到一个数据库的一致视图，需要注意的是，在只读事务中不允许进行任何写操作，只能进行查找操作或是复制数据库。提交和回滚只读事务的方式与前面介绍的提交和回滚读写事务的方式相同。

每次开启一个新的读写事务时，都要等待上一次读写事务结束，在提交读写事务的时候，会进行 I/O 操作将提交的修改写入磁盘。我们通过批处理读写事务，将并发的多个小事务合并

到一起，这样就可以减少读写事务的串行等待时间事务提交时等待磁盘带来的开销。开启批处理读写事务是通过调用 Batch()方法完成的，具体示例如下：

```
err := db.Batch(func(tx *bolt.Tx) error {
    // 批量事务中的相关操作
    return nil
})
```

这里需要特别注意的是，并发的批处理事务组合成更大的事务是随机的，所以只有当多个 goroutine 调用时，批处理事务才是有效的。另外，如果批处理事务中的部分操作失败了，则该批处理事务会尝试多次调用给定的函数，如果多次调用成功则整个事务成功，所以该函数必须具有幂等性。

上面介绍的 View()方法、Update()方法和 Batch()方法是 BoltDB 官网推荐的事务使用方式。这些方法是对 Begin()方法的封装，实际上它们底层会调用 Begin()方法启动事务，然后执行一个函数，最后安全地关闭事务（回滚或是提交）。但是在有些场景下，用户需要手动启动和结束事务，这种场景下我们可以直接调用 Begin()方法，具体示例如下：

```
tx, err := db.Begin(true) // 调用 Begin()方法开启一个事务，第一个参数表示开启的事务是否为读
                          // 写事务
if err != nil {
    return err
}
defer tx.Rollback() // 确保事务结束

_, err := tx.CreateBucket([]byte("MyBucket")) // 使用事务
if err != nil {
    return err
}

if err := tx.Commit(); err != nil { // 当操作完成之后，提交事务
    return err
}
```

通过上述示例代码可以看出，在 BoltDB 中手动管理事务与其他数据库类似。

了解了 BoltDB 的事务之后，下面介绍如何查询 BoltDB 中的数据。在 BoltDB 中，Bucket 表示的是数据库中键值对的集合，类似于 MySQL 中表的概念，在 Bucket 中的键类似于 MySQL 中的主键，必须是唯一的。通过 CreateBucket()方法可以创建一个 Bucket 实例，具体示例代码

如下：

```
db.Update(func(tx *bolt.Tx) error {
    b, err := tx.CreateBucket([]byte("TestBucket")) // 通过一个事务创建 Bucket 实例
    if err != nil {
        return fmt.Errorf("create bucket: %s", err)
    }
    return nil
})
```

除了 CreateBucket()方法，我们还可以使用 CreateBucketIfNotExists()方法在 Bucket 实例不存在的情况下才创建 Bucket 实例，其使用方式与 CreateBucket()方法类似。如果需要删除一个 Bucket 实例，则可以调用 DeleteBucket()方法完成删除。

要在 Bucket 实例中保存一个键值对，可以通过 Bucket.Put()函数完成，具体示例代码如下：

```
db.Update(func(tx *bolt.Tx) error {
    // 根据名称获取 Bucket 实例，如果对应的 Bucket 实例不存在，则返回 nil
    b := tx.Bucket([]byte("TestBucket"))
    err := b.Put([]byte("TestKey"), []byte("42")) // 调用 Put()方法保存键值对
    return err
})
```

要从 Bucket 实例中获取某个键值对时，可以通过 Bucket.Get()方法实现，具体示例代码如下：

```
db.View(func(tx *bolt.Tx) error {
    b := tx.Bucket([]byte("TestBucket")) // 获取 Bucket 实例
    // 调用 Get()方法获取键值对，如果对应的键值对不存在，则 Get()方法返回 nil
    v := b.Get([]byte("TestKey "))
    fmt.Printf("The TestKey is: %s\n", v)
    return nil
})
```

需要特别注意的是，从 Get()方法中返回的值只有在事务打开时才有效，如果需要在事务之外使用该值，则必须将其复制一份。

除了 Get()方法和 Put()方法，Bucket 还提供了 Delete()方法来删除一个键值对，其使用方式比较简单，这里不再展示其实例代码。

熟悉其他数据库产品（例如，MySQL、Oracle 等）的读者可能知道，在实际开发中经常会使用其自增主键作为主键。在 BoltDB 中可以通过 Bucket.NextSequence()方法提供一个自增序列，该序列可以作为混入键值对的唯一标识符，该方法的使用示例如下：

```
// CreateUser saves u to the store. The new user ID is set on u once the data is persisted.
func (s *Store) CreateUser(u *User) error {
    return s.db.Update(func(tx *bolt.Tx) error {
        b := tx.Bucket([]byte("users")) // 查询 users 对应的 Bucket 实例
        // 生成 User 的 id，当事务关闭时，NextSequence()方法会返回一个错误
        id, _ := b.NextSequence()
        u.ID = int(id)
        buf, err := json.Marshal(u) // 序列化 User 实例
        if err != nil {
            return err
        }
        return b.Put(u.ID, buf) // 调用 Bucket.Put()方法保存键值对
    })
}

type User struct {
    ID int
    ...
}
```

Bucket 中的 Key 是按顺序排列的，我们可以使用 Cursor 遍历 Bucket 中所有的键值对，该方法的使用示例如下：

```
db.View(func(tx *bolt.Tx) error {
    b := tx.Bucket([]byte("TestBucket")) // 查找对应的 Bucket 实例
    c := b.Cursor() // 创建 Cursor 实例
    for k, v := c.First(); k != nil; k, v = c.Next() { // 遍历 Bucket 中全部的键值对
        fmt.Printf("key=%s, value=%s\n", k, v)
    }
    return nil
})
```

BoltDB 中的 Curser 与其他数据库产品的游标很类似，但是它允许快速定位到某些指定的键值对或是前后移动，Curser 提供了如下方法。

- First()方法：移动到 Bucket 中的第一条键值对。
- Last()方法：移动到 Bucket 中的最后一条键值对。
- Seek()方法：移动到指定 Key 对应的键值对。
- Next()方法：移动到下一条键值对。
- Prev()方法：移动到上一条键值对。

这些函数都会返回对应键值对的信息（其中的 Key 和 Value 都是[]byte 类型），当迭代到末尾时，Next()方法返回的 Key 为 nil。正如上面实例展示的那样，在调用 Next()方法或 Prev()方法之前，必须使用 First()方法、Last()方法或 Seek()方法进行定位，如果不找到一个开始迭代的位置，那么 Next()方法和 Prev()方法将始终返回 nil。在迭代过程中，如果返回的 Key 不是 nil，而对应的 Value 为 nil，则意味着该 Key 在该 Bucket 中不存在对应的 Value。

前面提到，我们可以通过 Tx.Bucket()方法获取指定的 Bucket 实例。在 Bucket 中也提供了 Bucket()方法，该方法用来获取当前 Bucket 实例中的子 Bucket 实例，该方法的使用比较简单，这里不再举例说明。

在有些场景下，我们需要获取以某个字符串为前缀的所有键值对，类似于在 SQL 语句中的“like "xxx%"”的语句，此时就可以利用 Bucket 中键值对有序这一特性，通过 Seek()方法直接定位到前缀的位置，然后开始后续遍历，具体示例如下：

```
db.View(func(tx *bolt.Tx) error {
    c := tx.Bucket([]byte("TestBucket")).Cursor() // 获取指定的 Bucket 实例
    prefix := []byte("1234") // 前缀字符串

    // 通过 Seek()方法直接定位到前缀的位置，然后开始后续遍历
    for k, v := c.Seek(prefix); k != nil && bytes.HasPrefix(k, prefix); k, v = c.Next() {
        fmt.Printf("key=%s, value=%s\n", k, v)
    }
    return nil
})
```

另一个常见操作是范围扫描，例如，扫描一个给定的时间范围，具体示例代码如下：

```
db.View(func(tx *bolt.Tx) error {
    c := tx.Bucket([]byte("TestBucket ")).Cursor()
    // 下面是给定的扫描范围
    min := []byte("2016-01-01T00:00:00Z")
    max := []byte("2018-01-01T00:00:00Z")
```

```
    for k, v := c.Seek(min); k != nil && bytes.Compare(k, max) <= 0; k, v = c.Next() {
        fmt.Printf("%s: %s\n", k, v)
    }
    return nil
})
```

最后一个需要介绍的函数是 Bucket.ForEach()，其功能也是遍历给定 Bucket 实例中的所有键值对，具体示例实现如下所示。

```
db.View(func(tx *bolt.Tx) error {
    b := tx.Bucket([]byte("TestBucket"))
    b.ForEach(func(k, v []byte) error {
        fmt.Printf("key=%s, value=%s\n", k, v)
        return nil
    })
    return nil
})
```

BoltDB 作为一个单文件的数据库，备份整个数据库是十分简单的。我们可以使用 Tx.WriteTo()方法将数据库的一致视图写入指定的 Writer 中。 建议在一个只读事务中调用该方法完成备份，这样不会阻塞其他的读写操作。Tx.WriteTo()方法的使用比较简单，这里不再举例介绍。

通过本小节的介绍，我们简单了解了 BoltDB 的特性及其常用 API 接口。在实践中挑选合适的数据库，除了要了解数据库的优点，还要评估其限制及使用时的注意事项：

- BoltDB 在数据库文件上使用了独占的写锁，因此同一数据库不能被多个进程共享。
- BoltDB 适合读多写少的场景，顺序写性能也比较快速，但是随机写的速度会比较慢。
- BoltDB 在内部索引使用了 B+树的数据结构，因此会有较多的随机页面访问。在 SSD 磁盘上，BoltDB 的性能有显著提高。
- 尽量避免长期运行的事务。长时间执行的读写事务，会导致其他读写事务挂起等待。正如前面提到的，BoltDB 中使用写时复制技术，如果存在长时间的只读事务，则会导致脏页面不能被回收，因为可能有只读事务正在使用它们。
- 在查询数据时需要注意，BoltDB 返回的键值对数据只在事务期间有效。一旦事务被提交或回滚，那么它们指向的内存可以被重用，所以如果需要在事务之外使用，则需要对数据进行复制。
- 对于具有随机插入的 Bucket 使用时要小心，注意设置其页面利用率，防止其中出现大

量利用率较低的页面。

- 批量加载大量随机写入一个 Bucket 中可能会很慢，因为在事务提交之前，页面不会分裂。不建议在单个事务中随机插入超过 100000 个键值对到单个新 bucket 中。
- BoltDB 底层使用操作系统提供的 mmap 技术映射数据库文件，因此底层操作系统会处理进行缓存。通常，操作系统会在内存中缓存尽可能多的文件，这意味着数据库较大时，BoltDB 会占用很大的内存，同时操作系统将根据需要释放内存。只要它的内存映射适合于进程虚拟地址空间，BoltDB 可以处理比可用物理 RAM 大得多的数据，故不建议使用 32 位操作系统。

2. backend store 实现

在开始介绍 backend store 的具体实现之前，先介绍 etcd 是如何使用 BoltDB 实现数据存储的，同时会简单介绍 BoltDB 中保存的键值对数据的大致结构。etcd 在 BoltDB 中存储的 Key 是 reversion，Value 是 etcd 自定义的键值对组合。也就是说，etcd 会将键值对的每个版本都保存到 BoltDB 中，这也是 etcd 实现多版本机制的基础。

下面通过一个示例简单说明一下 etcd 的多版本机制，假设现在我们通过批量接口写入了两个键值对，分别是（key1，value1）和（key2，value2），之后我们再调用批量更新接口更新这两个键值对，更新后为（key1，update1）和（key2，update2）。此时在 BoltDB 中实际上记录了如下四条数据：

```
rev={1 0}, key=key1, value="value1"
rev={1 1}, key=key2, value="value2"
rev={2 0}, key=key1, value="update1"
rev={2 1}, key=key2, value="update2"
```

其中，reversion 主要由两部分组成，第一部分是 main reversion，每次事务递增一；第二部分是 sub reversion，同一个事务中的每次操作都会递增 1，两者结合就可以保证 Key 唯一且递增。在上述示例中，第一个事务的 main reversion 是 1，第二个事务的 main reversion 是 2，在第一个事务中，两次操作的 sub reversion 分别是 0 和 1。etcd 提供了压缩相关的配置选项，可以定时清理陈旧键值对，同时为 Put 操作提供了相关参数，用于设置某个 Key 的历史版本数。

从 backend store 保存的数据格式我们可以看出，如果要从 BoltDB 中查询键值对，必须通过 reversion 进行查找。但客户端只知道具体的键值对中的 Key 值，并不清楚每个键值对对应的 reversion 信息，所以在 v3 版本存储的内存索引（kvIndex）中保存的就是 Key 与 reversion 之前的映射关系。索引相关的原理和实现在后面详细介绍，下面先分析 backend store 的具体实现。

这里首先需要介绍的是 Backend 接口，根据 v3 版本存储的设计，其底层的存储是可以进行

切换的。Backend 接口的主要功能就是将底层存储与上层进行解耦，其中定义底层存储需要对上层提供的接口，其核心方法的定义如下：

```
type Backend interface {
    // 创建一个只读事务，这里的 ReadTx 接口是 v3 存储对只读事务的抽象，后面会详细介绍其实现
    ReadTx() ReadTx

    // 创建一个批量事务，这里的 BatchTx 接口是对批量读写事务的抽象，后面会详细介绍其实现
    BatchTx() BatchTx

    // 创建快照，后面介绍的 backend.snapshot 实现了该 Snapshot 接口
    Snapshot() Snapshot

    Size() int64    // 获取当前已存储的总字节数

    Defrag() error  // 碎片整理，其具体实现在后面详细介绍

    ForceCommit()  // 提交批量读写事务
}
```

结构体 backend 是 v3 版本存储提供的 Backend 接口的默认实现，其底层存储使用的就是上面介绍的 BoltDB。下面先来介绍一下其核心字段的含义。

- db（*bolt.DB 类型）：底层的 BoltDB 存储。
- size（int64 类型）：当前 backend 实例已存储的总字节数。
- commits（int64 类型）：从启动到目前为止，已经提交的事务数。
- batchInterval（time.Duration 类型）：两次批量读写事务提交的最大时间差。
- batchLimit（int 类型）：指定一次批量事务中最大的操作数，当超过该阈值时，当前的批量事务会自动提交。
- batchTx（*batchTxBuffered 类型）：批量读写事务，batchTxBuffered 是在 batchTx 的基础上添加了缓存功能，两者都实现了前面提到的 BatchTx 接口，batchTx 和 batchTxBuffered 的具体实现会在后面会详细介绍。
- readTx（*readTx 类型）：只读事务，readTx 实现了前面提到的 ReadTx 接口。

在 backend 中涉及另外两个比较重要的接口——ReadTx 和 BatchTx，它们分别是 etcd 对只读事务和批量读写事务的抽象。我们先来看一下 ReadTx 接口的定义：

```
type ReadTx interface {
  // 省略加锁解锁的方法

  // 在指定的 Bucket 中进行范围查找
  UnsafeRange(bucketName []byte, key, endKey []byte, limit int64)
      (keys [][]byte, vals [][]byte)

  // 遍历指定 Bucket 中的全部键值对
  UnsafeForEach(bucketName []byte, visitor func(k, v []byte) error) error
}
```

接下来看一下 BatchTx 接口的定义：

```
type BatchTx interface {
  ReadTx // 内嵌了 ReadTx 接口

  UnsafeCreateBucket(name []byte) // 创建 Bucket

  UnsafePut(bucketName []byte, key []byte, value []byte) // 向指定 Bucket 中添加键值对

  // 向指定 Bucket 中添加键值对，与 UnsafePut()方法的区别是，其中会将对应 Bucket 实例的
  // 填充比例设置为 90%，这样可以在顺序写入时，提高 Bucket 的利用率
  UnsafeSeqPut(bucketName []byte, key []byte, value []byte)

  UnsafeDelete(bucketName []byte, key []byte) // 在指定 Bucket 中删除指定的键值对

  Commit() // 提交当前的读写事务，之后立即打开一个新的读写事务

  CommitAndStop() // 提交当前的读写事务，之后并不会再打开新的读写事务
}
```

了解完这些重要的接口之后，我们再回到 backend 结构体来分析其提供的功能。首先来看 NewDefaultBackend()和 newBackend()，这两个函数都是用来初始化 backend 实例的，前者使用了默认的配置参数并且其底层是调用后者实现的，这些配置参数被封装成了 BackendConfig 结构体，其中各个字段的含义如下。

- Path（string 类型）：BoltDB 数据库文件的路径。
- BatchInterval（time.Duration 类型）：提交两次批量事务的最大时间差，用来初始化

backend 实例中的 batchInterval 字段，默认值是 100ms。

- BatchLimit（int 类型）：指定每个批量读写事务能包含的最多的操作个数，当超过这个阈值之后，当前批量读写事务会自动提交。该字段用来初始化 backend 中的 batchLimit 字段，默认值是 10000。
- MmapSize（uint64 类型）：在前面提到，BoltDB 使用了 mmap 技术对数据库文件进行映射，该字段用来设置 mmap 中使用的内存大小，该字段会在创建 BoltDB 实例时使用。

接下来，我们深入分析 newBackend()函数的具体实现。

```
func newBackend(bcfg BackendConfig) *backend {
    bopts := &bolt.Options{} // 初始化 BoltDB 时的参数
    if boltOpenOptions != nil {
        *bopts = *boltOpenOptions
    }
    bopts.InitialMmapSize = bcfg.mmapSize() // mmap 使用的内存大小
    db, err := bolt.Open(bcfg.Path, 0600, bopts) // 创建 bolt.DB 实例，省略错误处理
    b := &backend{ // 创建 backend 实例，并初始化其中各个字段
        db: db,
        batchInterval: bcfg.BatchInterval,
        batchLimit:    bcfg.BatchLimit,
        // 创建 readTx 实例并初始化 backend.readTx 字段
        readTx: &readTx{buf: txReadBuffer{
            txBuffer: txBuffer{make(map[string]*bucketBuffer)}},
        },
        ... ...
    }
    b.batchTx = newBatchTxBuffered(b) // 创建 batchTxBuffered 实例并初始化 backend.batchTx 字段
    go b.run() // 启动一个单独的 goroutine，其中会定时提交当前的批量读写事务，并开启新的批量读写事务
    return b
}
```

在 backend.run()方法中，会按照 batchInterval 指定的时间间隔，定时提交批量读写数据，在提交之后会立即开启一个新的批量读写事务。backend.run()方法的具体实现如下：

```
func (b *backend) run() {
    defer close(b.donec)
    t := time.NewTimer(b.batchInterval) // 创建定时器
```

```
    defer t.Stop()
    for {
        select { // 阻塞等待上述定时器到期
        case <-t.C:
        case <-b.stopc:
            b.batchTx.CommitAndStop()
            return
        }
        b.batchTx.Commit() // 提交当前的批量读写事务，并开启一个新的批量读写事务
        t.Reset(b.batchInterval)  // 重置定时器
    }
}
```

backend.ReadTx()方法和 BatchTx()方法分别返回了 backend 中的 readTx 字段和 batchTx 字段。backend.ForceCommit()方法会提交当前的读写事务并立即开启新的读写事务。backend.Size()方法返回的是 backend.size 字段值。backend.Commits()方法返回的是 backend.commits 字段。这些方法的实现比较简单，这里不再贴出具体代码。

backend.Snapshot()方法的主要功能是用当前的 BoltDB 中的数据创建相应的快照，其中使用前面提到的 Tx.WriteTo()方法备份整个 BoltDB 数据库的数据，具体实现如下：

```
func (b *backend) Snapshot() Snapshot {
    b.batchTx.Commit() // 提交当前的读写事务，主要是为了提交缓冲区中的操作
    // 省略加锁和解锁的相关操作
    tx, err := b.db.Begin(false) // 开启一个只读事务，错误处理(略)

    stopc, donec := make(chan struct{}), make(chan struct{})
    dbBytes := tx.Size() // 获取整个 BoltDB 中保存的数据

    go func() {  // 启动一个单独的 goroutine，用来检测快照数据是否已经发送完成
        defer close(donec)
        var sendRateBytes int64 = 100 * 1024 * 1014 // 这里假设发送快照的最小速度是 100MB/s
        // 创建定时器
        warningTimeout := time.Duration(int64((float64(dbBytes)
              / float64(sendRateBytes)) * float64(time.Second)))
        start := time.Now()
        ticker := time.NewTicker(warningTimeout)
        defer ticker.Stop()
```

```
        for {
            select {
            case <-ticker.C: // 超时未发送完快照数据，则会输出警告日志
                plog.Warningf("...")
            case <-stopc: // 发送快照数据结束
                snapshotDurations.Observe(time.Since(start).Seconds())
                return
            }
        }
    }()
    return &snapshot{tx, stopc, donec} // 创建快照实例
}
```

这里简单介绍一下 backend.Snapshot()方法返回的 backend.snapshot，该结构体实现了 backend.Snapshot 接口，该接口中最重要的接口就是 WriteTo()方法。backend.snapshot 中内嵌了 bolt.Tx，backend.snapshot.WriteTo()方法就是通过 Tx.WriteTo()方法实现的。

通过名称也能看出，backend.Defrag()方法的主要功能就是整理当前 BoltDB 实例中的碎片，其实就是提高其中 Bucket 的填充率。整理碎片实际上是创建新的 BoltDB 数据库文件并将旧数据库文件的数据写入新数据库文件中。因为在写入新数据库文件时是顺序写入的，所以会提高填充比例（FillPercent），从而达到整理碎片的目的。需要注意的是，在整理碎片的过程中，需要持有 readTx、batchTx 和 backend 的锁。下面来看一下 backend.Defrag()方法的具体实现：

```
func (b *backend) Defrag() error {
    err := b.defrag() // 调用 defrag()方法完成具体的碎片整理过程
    b.batchTx.Commit() // 碎片整理完成之后，立即提交一次事务
    return nil
}
// 下面是 backend.defrag()方法的具体实现，其中主要完成了新老数据库文件的切换
func (b *backend) defrag() error {
    // 加锁，这里会获取 readTx、batchTx 和 backend 中的三把锁(略)
    // 提交当前的批量读写事务，注意参数，此次提交后不会立即打开新的批量读写事务
    b.batchTx.unsafeCommit(true)
    b.batchTx.tx = nil
    // 创建新的 bolt.DB 实例，对应的数据库文件是个临时文件
    tmpdb, err := bolt.Open(b.db.Path()+".tmp", 0600, boltOpenOptions)

    // 进行碎片整理，其底层是创建一个新的 BoltDB 数据库文件并将当前数据库中的全部数据写入到其中，
```

```
    // 在写入过程中，会将新 Bucket 的填充比例设置成 90%，从而达到碎片整理的效果
    err = defragdb(b.db, tmpdb, defragLimit)
    // 省略错误处理的相关代码

    dbp := b.db.Path() // 获取旧数据库文件的路径
    tdbp := tmpdb.Path()// 获取新数据库文件的路径

    err = b.db.Close() // 关闭旧的 bolt.DB 实例
    err = tmpdb.Close() // 关闭新的 bolt.DB 实例
    err = os.Rename(tdbp, dbp) // 重命名新数据库文件，覆盖旧数据库文件

    // 重新创建 bolt.DB 实例，此时使用的数据库文件是整理之后的新数据库文件
    b.db, err = bolt.Open(dbp, 0600, boltOpenOptions)
    // 开启新的批量读写事务以及只读事务
    b.batchTx.tx, err = b.db.Begin(true)

    b.readTx.buf.reset()
    b.readTx.tx = b.unsafeBegin(false)
    atomic.StoreInt64(&b.size, b.readTx.tx.Size())

    return nil
}
```

backend.defrag()的重载方法主要完成了从旧数据库文件向新数据库文件的复制键值对的功能，具体实现如下：

```
func defragdb(odb, tmpdb *bolt.DB, limit int) error {
    tmptx, err := tmpdb.Begin(true) // 在新数据库实例上开启一个读写事务，错误处理(略)
    tx, err := odb.Begin(false) // 在旧数据库实例上开启一个只读事务，错误处理(略)
    defer tx.Rollback() // 方法结束时关闭该只读事务
    c := tx.Cursor()  // 获取旧实例上的 Cursor，用于遍历其中的所有 Bucket
    count := 0
    for next, _ := c.First(); next != nil; next, _ = c.Next() {
        // 读取旧数据库实例中的所有 Bucket，并在新数据库实例上创建对应 Bucket
        b := tx.Bucket(next) // 获取指定的 Bucket 实例，错误处理(略)
        tmpb, berr := tmptx.CreateBucketIfNotExists(next)
        // 为提高利用率，将填充比例设置成 90%，因为下面会从读取旧 Bucket 中全部的键值对，
        // 并填充到新 Bucket 中，这个过程是顺序写入的
```

```
            tmpb.FillPercent = 0.9
            b.ForEach(func(k, v []byte) error { // 遍历旧 Bucket 中的全部键值对
                count++
                if count > limit { // 当读取的键值对数量超过阈值，则提交当前读写事务(新数据库)
                    err = tmptx.Commit()
                    if err != nil {
                        return err
                    }
                    // 重新开启一个读写事务，继续后面的写入操作，错误处理(略)
                    tmptx, err = tmpdb.Begin(true)
                    tmpb = tmptx.Bucket(next)
                    tmpb.FillPercent = 0.9 // 设置填充比例
                    count = 0
                }
                return tmpb.Put(k, v) // 将读取到的键值对写入新数据库文件中
            })
        }
        return tmptx.Commit() // 最后提交读写事务(新数据库)
    }
```

➢ ReadTx

ReadTx 接口的定义在上一小节中已经详细介绍过了，这里不再赘述。本小节主要介绍 ReadTx 接口的具体实现——结构体 ReadTx。下面先介绍 ReadTx 结构体中各个字段的含义：

- buf(txReadBuffer 类型): 该 buffer 主要用来缓存 Bucket 与其中键值对集合的映射关系。
- mu (sync.RWMutex 类型): 在读写 buf 中的缓存区数据时，需要获取该锁进行同步。
- tx(*bolt.Tx 类型): 该 readTx 实例底层封装的 bolt.Tx 实例，即 BoltDB 层面的只读事务。
- txmu (sync.Mutex 类型): 在进行查询之前，需要获取该锁进行同步。

下面先分析 readTx 中实现的 UnsafeRange()方法，通过名称也可以推断出其主要功能就是进行范围查询。在该方法的实现中，我们只能对 safeRangeBucket（即名称为“key”的 Bucket，该 Bucket 中的 key 就是前面介绍的 revision，value 为键值对）进行真正的范围查询，对其他 Bucket 的查询只能返回单个键值对。readTx.UnsafeRange()方法的具体实现如下所示。

```
func (rt *readTx) UnsafeRange(bucketName, key, endKey []byte, limit int64)
        ([][]byte, [][]byte) {
    // 对非法的 limit 值进行重新设置(略)
    // 只有查询 safeRangeBucket(即名称为 key 的 Bucket)时，才是真正的范围查询，否则只能返回一个
    // 键值对
```

```
    if limit > 1 && !bytes.Equal(bucketName, safeRangeBucket) {
        panic("do not use unsafeRange on non-keys bucket")
    }
    // 首先从缓存中查询键值对
    keys, vals := rt.buf.Range(bucketName, key, endKey, limit)
    // 检测缓存返回的键值对数量是否达到 limit 的限制，如果达到 limit 指定的上限，则直接返回缓存的查
    // 询结果
    if int64(len(keys)) == limit {
        return keys, vals
    }
    rt.txmu.Lock() // 获取 txmu 锁
    // 通过 unsafeRange()函数从 BoltDB 中查询
    k2, v2, _ := unsafeRange(rt.tx, bucketName, key, endKey, limit-int64(len(keys)))
    rt.txmu.Unlock() // 释放 txmu 锁
    // 将查询缓存的结果与查询 BlotDB 的结果合并，然后返回
    return append(k2, keys...), append(v2, vals...)
}
```

这里需要介绍一下 readTx 和 batchTx 使用到的缓存，它们分别是 txReadBuffer 和 txWriteBuffer，两者都内嵌了结构体 txBuffer。下面先来看一下 txBuffer 结构体的大致实现，它只有一个字段（buckets，map[string]*bucketBuffer 类型），其中记录了 Bucket 名称与对应 bucketBuffer 的映射关系。在 bucketBuffer 中缓存了对应 Bucket 中的键值对数据。上述结构体之间的关系大致如图 6-3 所示。

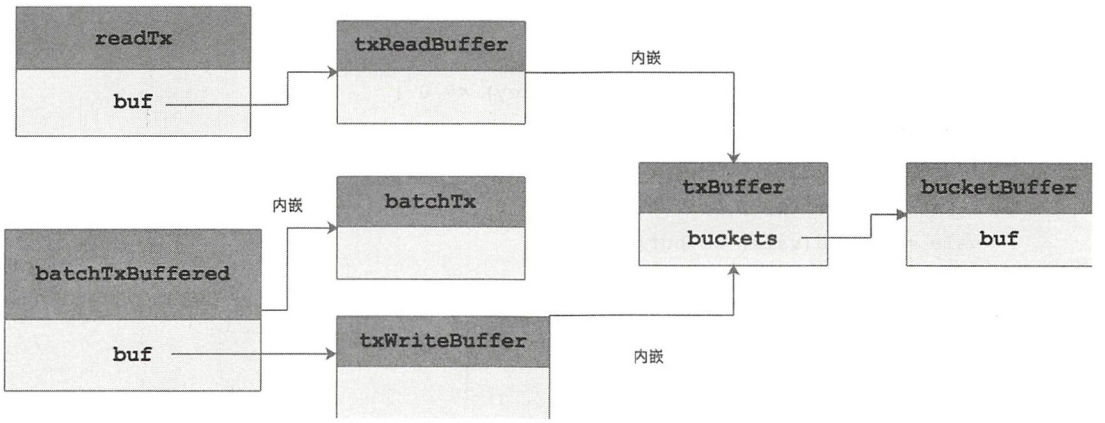


图 6-3

结构体 bucketBuffer 中各个字段的含义如下所示。

- **buf（[]kv 类型）**：每个元素都表示一个键值对，kv.key 和 kv.value 都是[]byte 类型。在初始化时，该切片的默认大小是 512。
- **used（int 类型）**：该字段记录 buf 中目前使用的下标位置。

下面来看一下 bucketBuffer 提供的方法，首先是 Range()方法，前面介绍的 readTx.Range()方法就会先调用该方法查找缓存，bucketBuffer.Range()方法的具体实现如下：

```
func (bb *bucketBuffer) Range(key, endKey []byte, limit int64)
        (keys [][]byte, vals [][]byte) {
  // 定义 Key 的比较方式
  f := func(i int) bool { return bytes.Compare(bb.buf[i].key, key) >= 0 }
  idx := sort.Search(bb.used, f) // 查询 0～used 之间是否有指定的 key
  if idx < 0 {
    return nil, nil
  }
  if len(endKey) == 0 { // 没有指定 endKey，则只返回 key 对应的键值对
    if bytes.Equal(key, bb.buf[idx].key) {
      keys = append(keys, bb.buf[idx].key)
      vals = append(vals, bb.buf[idx].val)
    }
    return keys, vals
  }
  // 如果指定了 endKey，则检测 endKey 的合法性(略)

  // 从前面查找到的 idx 位置开始遍历，直到遍历到 endKey 或是遍历的键值对个数达到 limit 上限为止
  for i := idx; i < bb.used && int64(len(keys)) < limit; i++ {
    if bytes.Compare(endKey, bb.buf[i].key) <= 0 {
      break
    }
    keys = append(keys, bb.buf[i].key)
    vals = append(vals, bb.buf[i].val)
  }
  return keys, vals // 返回全部符合条件的键值对
}
```

bucketBuffer.ForEach()方法提供了遍历当前 bucketBuffer 实例缓存的所有键值对的功能，其中会调用传入的 visitor()函数处理每个键值对，其具体实现如下所示。

```
func (bb *bucketBuffer) ForEach(visitor func(k, v []byte) error) error {
  for i := 0; i < bb.used; i++ { // 遍历 used 之前的所有元素
    // 调用 visitor()函数处理键值对，错误处理(略)
    if err := visitor(bb.buf[i].key, bb.buf[i].val); ...
  }
  return nil
}
```

bucketBuffer.add()方法提供了添加键值对缓存的功能，当 buf 的空间被用尽时，会进行扩容，具体实现如下所示。

```
func (bb *bucketBuffer) add(k, v []byte) {
  bb.buf[bb.used].key, bb.buf[bb.used].val = k, v // 添加键值对
  bb.used++ // 递增 used
  if bb.used == len(bb.buf) { // 当 buf 空间被用尽时，对其进行扩容
    buf := make([]kv, (3*len(bb.buf))/2)
    copy(buf, bb.buf)
    bb.buf = buf
  }
}
```

bucketBuffer.merge()方法主要负责将传入的 bbsrc（bucketBuffer 实例）与当前的 bucketBuffer 进行合并，之后会对合并结果进行排序和去重，具体实现如下所示。

```
func (bb *bucketBuffer) merge(bbsrc *bucketBuffer) {
  for i := 0; i < bbsrc.used; i++ { // 将 bbsrc 中的键值对添加到当前 bucketBuffer 中
    bb.add(bbsrc.buf[i].key, bbsrc.buf[i].val)
  }
  if bb.used == bbsrc.used { // 复制之前，如果当前 bucketBuffer 是空的，则复制完键值对之后
                             // 直接返回
    return
  }
  // 复制之前，如果当前 bucketBuffer 不是空的，则需要判断复制之后，是否需要进行排序
  if bytes.Compare(bb.buf[(bb.used-bbsrc.used)-1].key, bbsrc.buf[0].key) < 0 {
    return
  }
  // 如果需要排序，则调用 sort.Stable()函数对 bucketBuffer 进行排序
  // 注意，Stable()函数进行的是稳定排序，即相等键值对的相对位置在排序之后不会改变
```

```
    sort.Stable(bb)
    widx := 0
    for ridx := 1; ridx < bb.used; ridx++ { // 清除重复的 key，使用 key 的最新值
      if !bytes.Equal(bb.buf[ridx].key, bb.buf[widx].key) {
        widx++
      }
      bb.buf[widx] = bb.buf[ridx] // 新添加的的键值对覆盖原有的键值对
    }
    bb.used = widx + 1
}
```

介绍完 bucketBuffer 之后，我们回头来看一下 txBuffer，它只提供了一个 reset()方法，该方法负责清空 buckets 字段中的全部内容，其具体实现如下：

```
func (txb *txBuffer) reset() {
    for k, v := range txb.buckets { // 遍历 buckets
      if v.used == 0 { // 删除未使用的 bucketBuffer
        delete(txb.buckets, k)
      }
      v.used = 0 // 清空使用过的 bucketBuffer
    }
}
```

readTx 中的缓存（buf 字段）的类型是 txReadBuffer，在 txReadBuffer 中内嵌了 txBuffer。txReadBuffer 提供了 Range()和 ForEach()两个方法，这两个方法都是依赖 txBuffer 实现的，具体实现如下：

```
func (txr *txReadBuffer) Range(bucketName, key, endKey []byte, limit int64)
      ([][]byte, [][]byte) {
    if b := txr.buckets[string(bucketName)]; b != nil { // 查询指定的 txBuffer 实例
      return b.Range(key, endKey, limit) // 查询 txBuffer 实例
    }
    return nil, nil
}

func (txr *txReadBuffer) ForEach(bucketName []byte, visitor func(k, v []byte) error)
        error {
    if b := txr.buckets[string(bucketName)]; b != nil { // 查询指定的 txBuffer 实例
```

```
        return b.ForEach(visitor) // 调用 txBuffer.ForEach()方法完成遍历
    }
    return nil
}
```

在后面介绍的 batchTxBuffered 中使用的缓存（buf 字段）类型是 txWriteBuffer，在 txWriteBuffer 中内嵌了 txBuffer，另外 txWriteBuffer 中还有一个 seq 字段（bool 类型），用于标记写入当前 txWriteBuffer 的键值对是否为顺序的。下面看一下 txWriteBuffer.putSeq()方法，该方法完成了向指定 bucketBuffer 添加键值对的功能，具体实现如下：

```
func (txw *txWriteBuffer) putSeq(bucket, k, v []byte) {
    b, ok := txw.buckets[string(bucket)] // 获取指定的 bucketBuffer
    if !ok { // 如果未查找到，则创建对应的 bucketBuffer 实例并保存到 buckets 中
        b = newBucketBuffer()
        txw.buckets[string(bucket)] = b
    }
    b.add(k, v) // 通过 bucketBuffer.add()方法添加键值对
}
```

txWriteBuffer.put()方法底层调用了 putSeq()方法，同时将 seq 字段设置为 false，表示非顺序写入，具体实现如下：

```
func (txw *txWriteBuffer) put(bucket, k, v []byte) {
    txw.seq = false
    txw.putSeq(bucket, k, v)
}
```

最后，我们来看一下 txWriteBuffer.writeback()方法，该方法会将当前 txWriteBuffer 中键值对合并到指定的 txReadBuffer 实例中，这样就可以达到更新只读事务缓存的效果，其具体实现如下所示。

```
func (txw *txWriteBuffer) writeback(txr *txReadBuffer) {
    for k, wb := range txw.buckets { // 遍历所有的 bucketBuffer
        rb, ok := txr.buckets[k] // 从传入的 bucketBuffer 中查找指定的 bucketBuffer
        // 如果 txReadBuffer 中不存在对应的 bucketBuffer，则直接使用 txWriteBuffer 中缓存
        // 的 bucketBuffer 实例
        if !ok {
            delete(txw.buckets, k)
```

```
            txr.buckets[k] = wb
            continue
        }
        if !txw.seq && wb.used > 1 {
            sort.Sort(wb) // 如果当前 txWriteBuffer 中的键值对是非顺序写入的，则需要先进行排序
        }
        // 通过 bucketBuffer.merge()方法，合并两个 bucketBuffer 实例并去重，其具体实现已经详细
        // 介绍过了，这里不再赘述
        rb.merge(wb)
    }
    txw.reset() // 清空 txWriteBuffer
}
```

介绍完 txBuffer 的相关实现之后，读者应该清楚地了解了 readTx.unsafeRange()方法查询缓存的过程。下面介绍 unsafeRange()函数，readTx.unsafeRange()方法通过调用该函数完成对 BoltDB 的查询，具体实现如下：

```
func unsafeRange(tx *bolt.Tx, bucketName, key, endKey []byte, limit int64) (keys
[][]byte, vs [][]byte, err error) {
    bucket := tx.Bucket(bucketName) // 在 BoltDB 中查询指定的 Bucket
    if len(endKey) == 0 { // 如果没有指定 endKey，则直接查找指定 key 对应的键值对并返回
        if v := bucket.Get(key); v != nil {
            return append(keys, key), append(vs, v), nil
        }
        return nil, nil, nil
    }
    // 校正 limit 的值(略)
    c := bucket.Cursor() // 获取指定 Bucket 实例上的 Cursor 并从 key 位置开始遍历
    for ck, cv := c.Seek(key); ck != nil && bytes.Compare(ck, endKey) < 0; ck, cv = c.Next()
    {
        vs = append(vs, cv) // 记录符合条件的 value 值
        keys = append(keys, ck) // 记录符合条件的 key 值
        if limit == int64(len(keys)) {
            break
        }
    }
    return keys, vs, nil
}
```

接下来，我们看一下 readTx.UnsafeForEach()方法，该方法会遍历指定 Bucket 的缓存和 Bucket 中的全部键值对，并通过 visitor 回调函数处理这些遍历到的键值对，其具体实现如下：

```
func (rt *readTx) UnsafeForEach(bucketName []byte, visitor func(k, v []byte) error)
error {
    dups := make(map[string]struct{})
    f1 := func(k, v []byte) error { // f1 和 f2 两个函数是对 visitor 回调函数的一层包装
      dups[string(k)] = struct{}{}
      return visitor(k, v)
    }
    f2 := func(k, v []byte) error {
      if _, ok := dups[string(k)]; ok {
        return nil
      }
      return visitor(k, v)
    }
    // 调用 bucketBuffer.ForEach()方法，遍历缓存中的键值对。bucketBuffer.ForEach()方法的具
    // 体实现在前面介绍过了，这里不再赘述。错误处理(略)
    if err := rt.buf.ForEach(bucketName, f1); ...
    rt.txmu.Lock()   // 操作 readTx.tx 字段之前，需要获取 txmu 锁
    // 调用 unsafeForEach()函数，遍历 BoltDB 中的键值对
    err := unsafeForEach(rt.tx, bucketName, f2)
    rt.txmu.Unlock()
    return err
}
```

在 unsafeForEach()函数中，会调用前面介绍的 Bucket.ForEach()方法实现对 BoltDB 中键值对的遍历，具体实现如下：

```
func unsafeForEach(tx *bolt.Tx, bucket []byte, visitor func(k, v []byte) error) error {
    if b := tx.Bucket(bucket); b != nil {  // 查找指定的 Bucket 实例
      return b.ForEach(visitor) // 调用 Bucket.ForEach()方法进行遍历
    }
    return nil
}
```

➢ BatchTx

BatchTx 接口的定义在前面已经详细介绍过了，这里不再赘述。下面主要介绍 BatchTx 接口的具体实现——batchTx 和 batchTxBuffered。在前面提到过，batchTxBuffered 中内嵌了 batchTx 结构体，同时还包含一个 buf 字段（txWriteBuffer）作为缓存，在 backend 使用的是 batchTxBuffered 实例，而不是直接使用 batchTx 实例。另外，在 batchTx 结构体中内嵌了 sync.Mutex，并且在 batchTx 和 batchTxBuffered 中都重写了其 Unlock()方法。

下面介绍 batchTx 结构体中各个字段的含义。

- **tx（*bolt.Tx 类型）**：该 batchTx 实例底层封装的 bolt.Tx 实例，即 BoltDB 层面的读写事务。
- **backend（*backend 类型）**：该 batchTx 实例关联的 backend 实例。
- **pending（int 类型）**：当前事务中执行的修改操作个数，在当前读写事务提交时，该值会被重置为 0。

了解了 batchTx 和 batchTxBuffered 的结构之后，我们来看一下 batchTxBuffered 和 batchTx 的初始化过程，该过程是在 newBatchTxBuffered()函数中完成的，具体实现如下：

```
func newBatchTxBuffered(backend *backend) *batchTxBuffered {
  tx := &batchTxBuffered{
    batchTx: batchTx{backend: backend}, // 创建内嵌的 batchTx 实例
    buf: txWriteBuffer{ // 创建 txWriteBuffer 缓冲区
      txBuffer: txBuffer{make(map[string]*bucketBuffer)},
      seq:      true,
    },
  }
  tx.Commit() // 开启一个读写事务，下面会详细分析该方法的具体实现
  return tx
}
```

batchTxBuffered.Commit()方法是通过调用 batchTxBuffered.unsafeCommit(false)方法实现的，它会先回滚当前的只读事务，提交当前的读写事务，然后开启新的只读事务和读写事务。注意，在 Commit()方法中需要更新 readTx 和 batchTxBuffered 的字段（例如，tx 字段、buf 字段及其他的统计字段等），所以需要先获取 readTx 和 batchTxBuffered 的锁进行同步。batchTxBuffered.unsafeCommit()方法的具体实现如下：

```
func (t *batchTxBuffered) unsafeCommit(stop bool) {
  if t.backend.readTx.tx != nil {
```

```
        // 如果当前已经开启了只读事务，则将该事务回滚(BoltDB 中的只读事务只能回滚，无法提交)，错误
        // 处理(略)
        if err := t.backend.readTx.tx.Rollback();...
        t.backend.readTx.buf.reset() // 清空 readTx 中的缓存
        t.backend.readTx.tx = nil
    }
    t.batchTx.commit(stop) // 如果当前已经开启了读写事务，则将该事务提交，并创建新的读写事务
    if !stop { // 根据 stop 参数，决定事务开启新的只读事务
        t.backend.readTx.tx = t.backend.begin(false)
    }
}
```

开启新的只读事务或读写事务都是通过 backend.begin()方法实现的，该方法除了会开启事务（根据参数决定开启哪种类型的事务），还会更新 backend 中的相关字段，其具体实现如下：

```
func (b *backend) begin(write bool) *bolt.Tx {
    tx := b.unsafeBegin(write) // 开启事务
    atomic.StoreInt64(&b.size, tx.Size()) // 更新 backend.size 字段，记录了当前数据库的大小
    return tx
}

func (b *backend) unsafeBegin(write bool) *bolt.Tx {
    tx, err := b.db.Begin(write) // 调用了 BoltDB 的 API 开启事务，错误处理(略)
    return tx
}
```

接下来分析 batchTx.commit()方法是如何处理读写事务的，其具体实现如下所示。

```
func (t *batchTx) commit(stop bool) {// 省略加锁、解锁和错误处理的相关代码
    if t.tx != nil {
        if t.pending == 0 && !stop { // 当前读写事务中未进行任何修改操作，则无须开启新事务
            atomic.StoreInt64(&t.backend.size, t.tx.Size()) // 更新当前数据库大小
            return
        }
        err := t.tx.Commit() // 通过 BoltDB 提供的 API 提交当前读写事务
        atomic.AddInt64(&t.backend.commits, 1) // 递增 backend.commits 字段
        t.pending = 0 // 重置 pending 字段
    }
```

```
    if !stop {
        t.tx = t.backend.begin(true) // 开启新的读写事务
    }
}
```

介绍完初始化过程之后，再来看一下 batchTx.UnsafeCreateBucket()方法，其中是直接调用 BoltDB 的 API 创建了相应的 Bucket 实例，其具体实现比较简单，这里不再展开详述，感兴趣的读者可以参考源码进行学习。

结构体 batchTx 实现的 UnsafePut()方法和 UnsafeSeqPut()方法都是通过调用 batchTx.unsafePut()方法实现的，两者是通过 seq 参数进行区分的。batchTx.unsafePut()方法的具体实现如下：

```
func (t *batchTx) unsafePut(bucketName []byte, key []byte, value []byte, seq bool) {
    bucket := t.tx.Bucket(bucketName) // 通过 BoltDB 提供的 API 获取指定的 Bucket 实例，错误
                                      // 处理(略)
    if seq { // 如果是顺序写入，则将填充率设置成 90%
        bucket.FillPercent = 0.9
    }
    if err := bucket.Put(key, value); ...// 调用 BoltDB 提供的 API 写入键值对，错误处理(略)
    t.pending++ // 递增 pending
}
```

上述 UnsafePut*()方法和 batchTx.UnsafeDelete()方法在执行之前，先要获取 batchTxBuffered 的锁才能正确执行。batchTx.UnsafeDelete()方法先通过 BoltDB API 查找到对应的 Bucket 实例，然后通过其 Delete()方法删除键值对，其实现比较简单，这里不再贴出代码，感兴趣的读者可以参考相关代码进行学习。

batchTx 实现的 UnsafeRange()方法和 UnsafeForEach()方法分别是通过调用 unsafeRange()函数和 unsafeForEach()方法实现的，前面介绍的 readTx 中的对应方法时，已经分析了这两个函数的具体实现，这里不再赘述。

接下来看一下 batchTxBuffered 提供的方法，在 batchTxBuffered.UnsafePut()方法和 UnsafeSeqPut()方法中，除了调用 batchTx 中对应的方法将键值对写入 BoltDB，还会调用其 buf 字段（txWriteBuffer 类型）的方法，将键值对写入 batchTxBuffered 的缓存中，具体实现如下：

```
func (t *batchTxBuffered) UnsafePut(bucketName []byte, key []byte, value []byte) {
    t.batchTx.UnsafePut(bucketName, key, value) // 将键值对写入 BoltDB
```

```
    t.buf.put(bucketName, key, value) // 将键值对写入缓存
}

func (t *batchTxBuffered) UnsafeSeqPut(bucketName []byte, key []byte, value []byte) {
    t.batchTx.UnsafeSeqPut(bucketName, key, value) // 将键值对写入 BoltDB
    t.buf.putSeq(bucketName, key, value) // 将键值对写入缓存
}
```

最后，需要详细说明的是 batchTx 和 batchTxBuffered 的 Unlock()方法，在 batchTx 和 batchTxBuffered 中都重写了内嵌的 sync.Mutex 的 Unlock()方法（但是它们并没有重新 Lock()方法）。首先来看一下 batchTx.Unlock()方法的具体实现，如下所示。

```
func (t *batchTx) Unlock() {
    if t.pending >= t.backend.batchLimit { // 检测当前事务的修改操作数量是否达到上限
        t.commit(false) // 提交当前读写事务，并开启新的事务
    }
    t.Mutex.Unlock() // 释放锁
}
```

在 batchTxBuffered.Unlock()方法中，除了调用 batchTx.Unlock()方法实现解锁，还会将当前 batchTxBuffered 中缓存的键值对更新到 readTx 缓存中（即通过前面介绍的 txWriteBuffer.writeback()方法），从而实现只读事务的缓存更新。这也是前面 UnsafePut*()等方法在执行前必须获取锁的原因。batchTxBuffered.Unlock()方法的具体实现如下：

```
func (t *batchTxBuffered) Unlock() {
    if t.pending != 0 { // 检测当前读写事务中是否发生了修改操作
        t.buf.writeback(&t.backend.readTx.buf) // 更新 readTx 的缓存
        if t.pending >= t.backend.batchLimit {
            t.commit(false) // 如果当前事务的修改操作数量是否达到上限，则提交当前事务，开启新事务
        }
    }
    t.batchTx.Unlock() // 调用 batchTx.Unlock()方法完成解锁
}
```

至此，backend store 的相关实现就介绍完了，下面将介绍 backend 中内存索引的原理及相关实现。

3. backend 索引实现

在介绍 backend store 实现之前提到，它实现了多版本的机制，也就是说，键值对的每一次更新操作都被单独记录下来了。在 BoltDB 中存储的键值对中，key 实际上是 reversion（由 main reversion 和 sub reversion 两部分组成）。这样，要从 BoltDB 中检索数据就必须通过 reversion 完成，为了将客户端提供的原始键值对信息与 reversion 关联起来，backend 在内存索引实现中，为每个客户端提供的原始 Key 关联了一个 keyIndex 实例，其中维护了多版本信息。

从整体上来看，客户端在查找指定键值对时，会先通过内存中维护的 B 树索引（该 B 树索引中维护了原始 Key 值到 keyIndex 的映射关系）查找到对应的 keyIndex 实例，然后通过 keyIndex 查找到对应的 revision 信息（keyIndex 内维护了多个版本的 revision 信息），最后通过 revision 映射到磁盘中的 BoltDB 查找并返回真正的键值对数据。大致流程如图 6-4 所示。

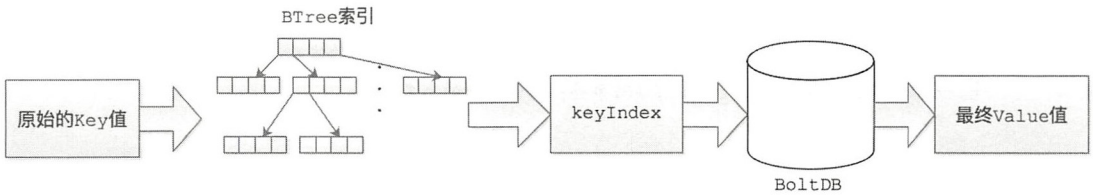


图 6-4

➢ keyIndex

下面详细介绍 keyIndex 中每个字段的含义。

- key（[]byte 类型）：客户端提供的原始 Key 值。
- modified（revision 类型）：记录该 Key 值最后一次修改对应的 revision 信息。在 revision 结构体中有如下两个字段。
  - main（int64 类型）：前面介绍的 main reversion。
  - sub（int64 类型）：前面介绍的 sub reversion。

  另外，在 revision 中还提供了排序、比较（GreaterThan()方法）、序列化等相关的方法，其实现都比较简单，这里不再赘述。
- generations（[]generations 类型）：当第一次创建客户端给定的 Key 值时，对应的第 0 代版本信息（即 generations[0]项）也会被创建，所以每个 Key 值至少对应一个 generation 实例（如果没有，则表示当前 Key 值应该被删除），每代中包含多个 reversion 信息。当客户端后续不断修改该 Key 时，generation[0]中会不断追加 revision 信息，如图 6-5 所示。

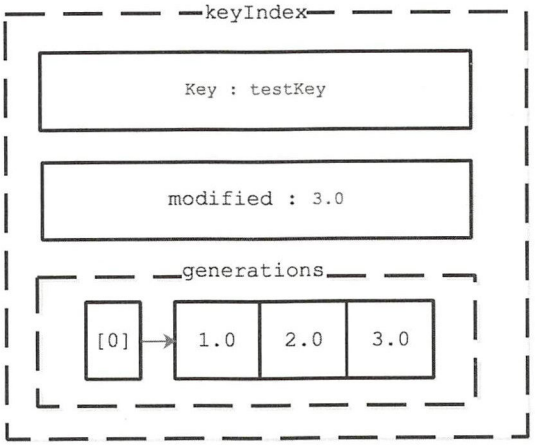


图 6-5

当向 generation 实例中追加一个 Tombstone 时，表示删除当前 Key 值，此时就会结束当前的 generation，后续不再向该 generation 实例中追加 reversion 信息，同时会创建新的 generation 实例，如图 6-6 所示。读者可以认为 generation 对应了当前 Key 一次从创建到删除的生命周期。

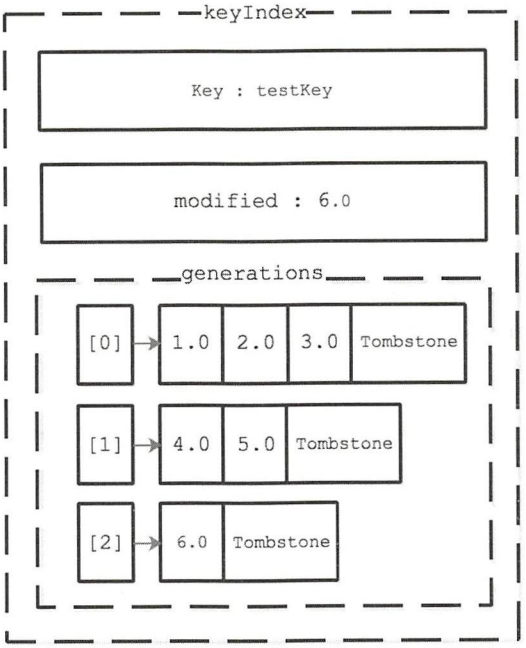


图 6-6



下面看一下 generation 结构体中各个字段的含义。

- created（revision 类型）：记录创建当前 generation 实例时对应的 revision 信息。
- revs（[]revision 类型）：当客户端不断更新该键值对时，revs 数组会不断追加每次更新对应 revision 信息。
- ver（int64 类型）：记录当前 generation 所包含的修改次数，即 revs 数组的长度。

了解了 keyIndex 中核心字段的含义之后，我们来看一下 keyIndex 中的提供的相关方法。首先来看 keyIndex.put()方法，该方法负责向 keyIndex 中追加新的 revision 信息，具体实现如下：

```
func (ki *keyIndex) put(main int64, sub int64) {
  // 根据传入的 main reversion 和 sub reversion 创建 reversion 实例
  rev := revision{main: main, sub: sub}
  // 检测该 reversion 实例的合法性(略)
  if len(ki.generations) == 0 { // 创建 generations[0]实例
    ki.generations = append(ki.generations, generation{})
  }
  g := &ki.generations[len(ki.generations)-1]
  if len(g.revs) == 0 { // 新建 Key，则初始化对应 generation 实例的 created 字段
    keysGauge.Inc()
    g.created = rev
  }
  g.revs = append(g.revs, rev) // 向 generation.revs 中追加 revision 信息
  g.ver++ // 递增 generation.ver
  ki.modified = rev // 更新 keyIndex.modified 字段
}
```

接下来看一下 keyIndex.tombstone()方法，该方法会在当前 generation 中追加一个 revision 实例，然后新建一个 generation 实例，具体实现如下：

```
func (ki *keyIndex) tombstone(main int64, sub int64) error {
  // 检测当前的 keyIndex.generations 字段是否为空，以及当前使用的 generation 实例是否为空(略)
  ki.put(main, sub)   // 调用上面介绍的 put()方法，在当前 generation 中追加一个 revision
  // 在 generations 中创建新 generation 实例
  ki.generations = append(ki.generations, generation{})
  return nil
}
```

keyIndex.get()方法会在当前 keyIndex 实例中查找小于指定的 main revision 的最大 revision，



具体实现如下：

```
func (ki *keyIndex) get(atRev int64) (modified, created revision, ver int64,err error){
    // 检测当前 keyInde 中是否记录了 generation 实例(略)
    // 根据给定的 main revision，查找对应的 generation 实例，如果没有对应的 generation，则报错
    g := ki.findGeneration(atRev)
    // 在 generation 中查找对应的 revision 实例，如果查找失败，就会报错
    n := g.walk(func(rev revision) bool { return rev.main > atRev })
    // 返回值除了目标 revision 实例，还有该 Key 此次创建的 revision 实例，以及到目标 revision 之前
    // 的修改次数
    return g.revs[n], g.created, g.ver - int64(len(g.revs)-n-1), nil
}
```

在查找指定 main revision 所在的 generation 实例时，keyIndex.get()方法是通过调用 keyIndex.findGeneration()方法实现的，findGeneration()方法的具体实现如下所示。

```
func (ki *keyIndex) findGeneration(rev int64) *generation {
    lastg := len(ki.generations) - 1
    cg := lastg // 指向当前 keyIndex 实例中最后一个 generation 实例，并逐个向前查找

    for cg >= 0 {
        if len(ki.generations[cg].revs) == 0 { // 过滤调用空的 generation 实例
            cg--
            continue
        }
        g := ki.generations[cg]
        if cg != lastg { // 如果不是最后一个 generation 实例，则先与 tombone revision 进行比较
            if tomb := g.revs[len(g.revs)-1].main; tomb <= rev {
                return nil
            }
        }
        if g.revs[0].main <= rev { // 与 generation 中的第一个 revision 比较
            return &ki.generations[cg]
        }
        cg--
    }
    return nil
}
```



有时 keyIndex.findGeneration()方法无法查询到指定的 main revision 所在的 generation 实例，假设当前的 main revision 是 5，当前 keyIndex.generations 的状态如图 6-7 所示，因为 main revision 为 4～7 这段时间内，该 Key 被删除了，所以无法查找到其所在的 generation 实例，此时 findGeneration()方法会返回 nil。

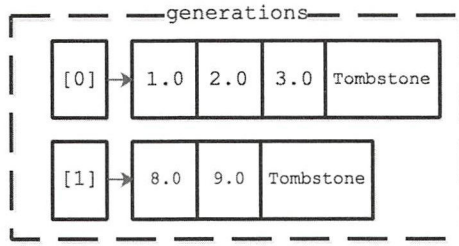


图 6-7

在查找到指定 main revision 所在的 generation 实例之后，会调用 generation.walk()方法查找从该 generation 实例中查找符合条件的 revision 实例，其具体实现如下：

```
func (g *generation) walk(f func(rev revision) bool) int {
  l := len(g.revs)
  for i := range g.revs {
    ok := f(g.revs[l-i-1]) // 逆序查找 generation.revs 数组
    if !ok {
      return l - i - 1  // 返回目标 revision 的下标
    }
  }
  return -1
}
```

keyIndex 结构体还提供了 since()方法用于批量查找 revision，该方法负责返回当前 keyIndex 实例中 main 部分大于指定值的 revision 实例，如果查询结果中包含了多个 main 部分相同的 revision 实例，则只返回其中 sub 部分最大的实例。keyIndex.since()方法的具体实现如下：

```
func (ki *keyIndex) since(rev int64) []revision {
  since := revision{rev, 0}
  var gi int
  // 倒序遍历所有 generation 实例，查询从哪个 generation 开始查找(即 gi 对应的 generation 实例)
  for gi = len(ki.generations) - 1; gi > 0; gi-- {
    g := ki.generations[gi]
```



```
        if g.isEmpty() {
            continue
        }
        // 比较创建当前 generation 实例的 revision 与 since
        // 在 generation.GreaterThan()方法中会先比较 main revision 部分，如果相同，
        // 则比较 sub revision 部分
        if since.GreaterThan(g.created) {
            break
        }
    }

    var revs []revision // 记录返回结果
    var last int64 // 用于记录当前所遇到的最大的 main revision 值
    for ; gi < len(ki.generations); gi++ { // 从 gi 处开始遍历 generation 实例
        for _, r := range ki.generations[gi].revs { // 遍历每个 generation 中记录的 revision
                                                    // 实例
            if since.GreaterThan(r) { // 忽略 main revision 部分较小的 revision 实例
                continue
            }
            // 如果查找到 mainrevision 部分相同且 sub revision 更大的 revision 实例，
            // 则用其替换之前记录的返回结果，从而实现 main 部分相同时只返回 sub 部分较大的 revision 实例
            if r.main == last {
                revs[len(revs)-1] = r
                continue
            }
            revs = append(revs, r) // 将符合条件的 revision 实例记录到 revs 中
            last = r.main // 更新 last
        }
    }
    return revs
}
```

随着客户端不断修改键值对，keyIndex 中记录的 revision 实例和 generation 实例会不断增加，我们可以通过调用 compact()方法对 keyIndex 进行压缩。在压缩时会将 main 部分小于指定值的全部 revision 实例全部删除。在压缩过程中，如果出现了空的 generation 实例，则会将其删除。如果 keyIndx 中全部的 generation 实例都被清除了，则该 keyIndex 实例也会被删除。这里通过一个简单的实例介绍一下压缩 keyIndex 的原理（compact(n)表示压缩掉 revision.main≤n 的所有



版本），假设当前 keyIndex 中 generations 字段的结构如图 6-8（A）所示。假设之后调用 compact(4)对当前 keyIndex 实例进行压缩，得到的结果如图 6-8（B）所示。假设之后又调用 compact(5)对当前 keyIndex 实例进行压缩，得到的结果如图 6-8（C）所示。此时，generations 数组为空，则该 keyIndex 实例也应该被删除。

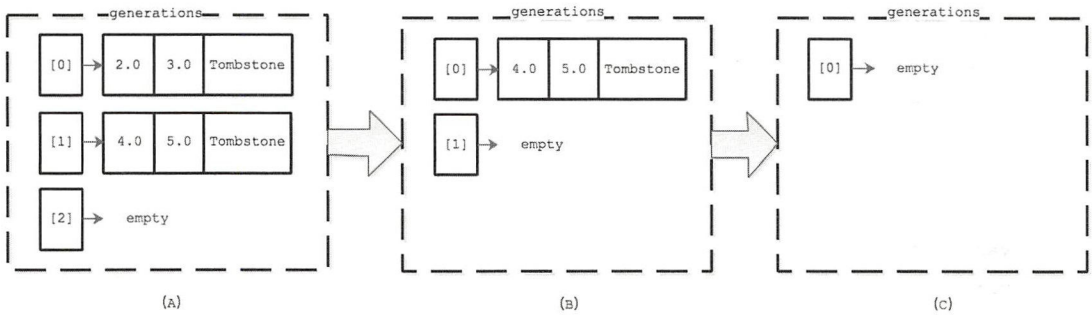


图 6-8

keyIndex.compact()方法的具体实现如下：

```
func (ki *keyIndex) compact(atRev int64, available map[revision]struct{}) {
    f := func(rev revision) bool { // 后面遍历 generation 时使用的回调函数
        if rev.main <= atRev {
            available[rev] = struct{}{}
            return false
        }
        return true
    }

    i, g := 0, &ki.generations[0]
    // 遍历所有的 generation 实例，目标 generation 实例中的 tombstone 大于指定的 revision
    for i < len(ki.generations)-1 {
        if tomb := g.revs[len(g.revs)-1].main; tomb > atRev {
            break
        }
        i++
        g = &ki.generations[i]
    }
    // 遍历目标 generation 中的全部 revision 实例，清空目标 generation 实例中 main 部分小于指
    // 定值的 revision
    if !g.isEmpty() {
```



```
        n := g.walk(f)
        if n != -1 {
            g.revs = g.revs[n:] // 清理目标 generation
        }
        // 如果目标 generation 实例中只有 tombstone，则将其删除
        if len(g.revs) == 1 && i != len(ki.generations)-1 {
            delete(available, g.revs[0])
            i++  // 递增 i，后面会清理 i 之前的全部 generation 实例
        }
    }
    ki.generations = ki.generations[i:] // 清理目标 generation 实例之前的全部 generation
                                        // 实例
}
```

keyIndex.restore()方法负责恢复当前的 keyIndx 中的信息，具体实现如下：

```
func (ki *keyIndex) restore(created, modified revision, ver int64) {
    ki.modified = modified
    // 创建 generation 实例，并添加到 generations 中
    g := generation{created: created, ver: ver, revs: []revision{modified}}
    ki.generations = append(ki.generations, g)
}
```

➢ treeIndex

etcd v3 版本存储在内存中维护的 BTree 索引（在很多其他的数据库产品中都可以看到 BTree 索引的身影）实际上是使用了 Google 开源的一个 Go 语言 BTree 实现（http://godoc.org/github.com/google/btree）。本小节后续将会简单介绍该 BTree 实现的相关 API，至于 BTree 的原理及具体实现不打算进行详细分析，需要了解这部分内容的读者请查阅相关文档。

下面简单介绍一下后续会涉及的 BTree API 的含义。首先在 Google 开源的 BTree 实现中，我们可以通过调用 btree.New()函数来创建 BTree 实例。在 BTree 中提供了 Ascend*方法用于正序遍历 BTree 中的元素，其中 Ascend(iterator ItemIterator)方法会正序遍历并处理当前 BTree 中的所有元素，AscendGreaterOrEqual(pivot Item, iterator ItemIterator)方法只会处理比 pivot 大的元素（正序），AscendLessThan(pivot Item, iterator ItemIterator)方法只会处理比 pivot 小的元素（正序），AscendRange(greaterOrEqual, lessThan Item, iterator ItemIterator)方法只会处理 greaterOrEqual 到 lessThan 之间的元素（正序）。BTree 还提供了 Descend*()方法用于逆序处理 BTree 中的元素，其含义与 Ascend*()方法类似，这里不再赘述。



BTree.ReplaceOrInsert(item Item)方法主要负责向该 BTree 中添加元素并返回 nil，如果待添加的元素已存在，则将其删除并返回。BTree.Get()方法和 Has()方法分别用来获取当前 BTree 实例中的指定元素，以及检测其中是否包含指定的元素。

BTree.Delete*()方法用于删除当前 BTree 实例中的元素，其中 Delete(item Item)方法会删除参数指定的元素，DeleteMax()方法、DeleteMin()方法分别删除该 BTree 中的最大元素和最小元素。BTree.Clear()方法会清空当前 BTree 中的全部元素，该方法会根据参数决定是否需要将所有元素先添加到 FreeList 中保存。最后，BTree.Len()、Max()和 Min()方法分别获取该 BTree 中的元素个数、最大元素和最小元素。

在 etcd 中，对 Google 开源的 BTree 实现进行了一层封装，对外提供了 index 接口，该接口的定义如下：

```
type index interface {
    // 查询指定的 key
    Get(key []byte, atRev int64) (rev, created revision, ver int64, err error)
    // 添加元素
    Put(key []byte, rev revision)
    Insert(ki *keyIndex)
    // 范围查询
    Range(key, end []byte, atRev int64) ([][]byte, []revision)
    RangeSince(key, end []byte, rev int64) []revision
    Tombstone(key []byte, rev revision) error // 添加 Tombstone
    Compact(rev int64) map[revision]struct{} // 压缩 BTree 中的全部 keyIndex
}
```

结构体 treeIndex 是 v3 版本存储提供的 index 接口实现，其中内嵌了 sync.RWMutex，在进行更新操作时，例如，Insert()、Compact()方法中，都需要获取该锁。在 treeIndex 中最重要的就是 tree 字段（*btree.BTree），通过该字段可以操作前面介绍的开源 BTree 实现。在 newTreeIndex()函数中完成了对 tree 字段的初始化，具体实现如下：

```
func newTreeIndex() index {
   return &treeIndex{
      // 这里将 BTree 的度初始化为 32，即除了根节点的每个节点至少有 32 个元素，每个节点最多有 64 个元素
      tree: btree.New(32),
   }
}
```

下面看一下 treeIndex.Put()方法，其中主要完成两项操作，一是向 BTree 中添加 keyIndex 实

例，二是向 keyIndex 中追加 revision 信息。Put()方法的具体实现如下：

```
func (ti *treeIndex) Put(key []byte, rev revision) {
   keyi := &keyIndex{key: key} // 创建 keyIndex 实例
   // 加锁和解锁的相关逻辑(略)
   // 通过 BTree.Get()方法在 BTree 上查找指定的元素
   item := ti.tree.Get(keyi)
   if item == nil { // 在 BTree 中不存在指定的 Key
      // 向 keyIndex 中追加一个 revision，keyIndex.put()方法在前面已经详细介绍过了，这里不再赘述
      keyi.put(rev.main, rev.sub)
      // 通过 BTree.ReplaceOrInsert()方法向 BTree 中添加 keyIndex 实例
      ti.tree.ReplaceOrInsert(keyi)
      return
   }
   // 如果在 BTree 中查找到该 Key 对应的元素，则向其中追加一个 revision 实例
   okeyi := item.(*keyIndex)
   okeyi.put(rev.main, rev.sub)
}
```

在 treeIndex 中还有一个 Insert()方法，它直接调用 BTree.ReplaceOrInsert()方法插入 keyIndex 实例，其实现比较简单，这里不再赘述。

介绍完向 BTree 添加元素的逻辑，下面介绍 treeIndex.Get()方法，该方法负责从 BTree 中查询 revision 信息，具体实现如下：

```
func (ti *treeIndex) Get(key []byte, atRev int64)
        (modified, created revision, ver int64, err error) {
   keyi := &keyIndex{key: key} // 创建 keyIndex 实例
   // 加锁和解锁的相关逻辑(略)
   // 查询指定 Key 对应的 keyIndex 实例
   if keyi = ti.keyIndex(keyi); keyi == nil {
      return revision{}, revision{}, 0, ErrRevisionNotFound
   }
   return keyi.get(atRev) // 在查询到的 keyIndex 实例中，查找对应的 revision 信息
}
```

treeIndex.keyIndex()方法通过 BTree.Get()方法查询指定的元素，除了 treeIndex.Get()方法依赖于该方法，treeIndex.KeyIndex()方法也依赖于该方法，其具体实现如下：

```
func (ti *treeIndex) keyIndex(keyi *keyIndex) *keyIndex {
  if item := ti.tree.Get(keyi); item != nil {
    return item.(*keyIndex)
  }
  return nil
}
```

treeIndex 还提供了范围查询的功能，先来看其中的 Range()方法，该方法负责查询 BTree 中 key～end 之间元素中指定 revision 信息，具体实现如下：

```
func (ti *treeIndex) Range(key, end []byte, atRev int64)
        (keys [][]byte, revs []revision) {
  if end == nil {
    // 如果未指定 end 参数，则只查询 key 对应的 revision 信息，错误处理(略)
    rev, _, _, err := ti.Get(key, atRev)
    return [][]byte{key}, []revision{rev}
  }
  // 创建 key 及 end 对应的 keyIndex 实例
  keyi := &keyIndex{key: key}
  endi := &keyIndex{key: end}
  // 加锁解锁的相关逻辑(略)
  // 调用 BTree.AscendGreaterOrEqual()方法遍历比 key 大的元素
  ti.tree.AscendGreaterOrEqual(keyi, func(item btree.Item) bool {
    if len(endi.key) > 0 && !item.Less(endi) { // 忽略 end 之后的元素
      return false
    }
    curKeyi := item.(*keyIndex)
    // 从当前元素中查询指定的 revision 元素，并追加到 revs 中，错误处理(略)
    rev, _, _, err := curKeyi.get(atRev)
    revs = append(revs, rev)
    keys = append(keys, curKeyi.key) // 将当前 key 追加到 keys 中，等待返回
    return true
  })
  return keys, revs
}
```

再来看一下 treeIndex.RangeSince()方法，该方法会查询了 key～end 的元素，并从中查询 main revision 部分大于指定值的 revision 信息。注意，最终返回的 revision 实例不是按照 key 进

行排序的，而是按照 revision.GreaterThan()排序的。RangeSince()方法的具体实现如下：

```
func (ti *treeIndex) RangeSince(key, end []byte, rev int64) []revision {
  // 加锁和解锁的相关逻辑(略)
  keyi := &keyIndex{key: key} // 创建 key 对应的 keyIndex 实例
  if end == nil {
    // 如果未指定 end 参数，则只查询 key 对应的 revision 信息
    item := ti.tree.Get(keyi)
    keyi = item.(*keyIndex)
    // 调用 keyIndex.since()方法，获取 rev 之后的 revision 实例，
    // keyIndex.since()方法在前面介绍过了，这里不再赘述
    return keyi.since(rev)
  }
  endi := &keyIndex{key: end}
  var revs []revision
  // 调用 BTree.AscendGreaterOrEqual()方法遍历比 key 大的元素
  ti.tree.AscendGreaterOrEqual(keyi, func(item btree.Item) bool {
    if len(endi.key) > 0 && !item.Less(endi) { // 忽略大于 end 的元素
      return false
    }
    curKeyi := item.(*keyIndex)
    // 调用 keyIndex.since()方法，获取当前 keyIndex 实例中 rev 之后的 revision 实例
    revs = append(revs, curKeyi.since(rev)...)
    return true
  })
  sort.Sort(revisions(revs)) // 对 revs 中记录的 revision 实例进行排序，并返回
  return revs
}
```

介绍完 treeIndex 的读写方法之后，再来分析 treeIndex.Compact()方法，其中会遍历 BTree 中所有的 keyIndex 实例，并调用 keyIndex.compact()方法对其中的 generations 进行压缩，具体实现如下：

```
func (ti *treeIndex) Compact(rev int64) map[revision]struct{} {
  available := make(map[revision]struct{})
  var emptyki []*keyIndex
  // 加锁和解锁的相关逻辑(略)
  // 调用 BTree.Ascend()方法遍历其中的全部 keyIndex 实例，在遍历过程中，会将待删除的 keyIndex
```

```
    // 实例添加到 emptyki 中
    ti.tree.Ascend(compactIndex(rev, available, &emptyki))
    for _, ki := range emptyki {
      item := ti.tree.Delete(ki)
    }
    return available
}
```

在 keyIndex.compact()方法中完成了具体的压缩操作，并记录了 generations 为空的 keyIndex 实例（即待删除的 keyIndex），具体实现如下：

```
func compactIndex(rev int64, available map[revision]struct{}, emptyki *[]*keyIndex)
      func(i btree.Item) bool {
  return func(i btree.Item) bool {
    keyi := i.(*keyIndex)
    //  调用 keyIndex.compact()方法进行压缩，该方法的具体实现在前面详细介绍过了，这里不再赘述
    keyi.compact(rev, available)
    if keyi.isEmpty() { // 检测 keyIndex 实例中 generations 是否已为空
      *emptyki = append(*emptyki, keyi) // 将空的 keyIndex 实例追加到 emptyki 中等待删除
    }
    return true
  }
}
```

最后需要简单提一下的是 treeIndex.Tombstone()方法，该方法会先在 BTree 中查找指定的 keyIndex 实例，然后调用 keyIndex.tombstone()方法结束其当前 generation 实例并创建新的 generation 实例。Tombstone()方法的实现比较简单，这里不再赘述。

4. mvcc.store

前面介绍了底层 BoltDB 存储和内存 BTree 索引的原理和相关实现，下面主要看一下 mvcc.store 如何将这些组件组合成一个整体并对外提供完整的服务。在开始介绍之前，先来介绍涉及的几个核心接口，它们之间的结构关系如图 6-9 所示。

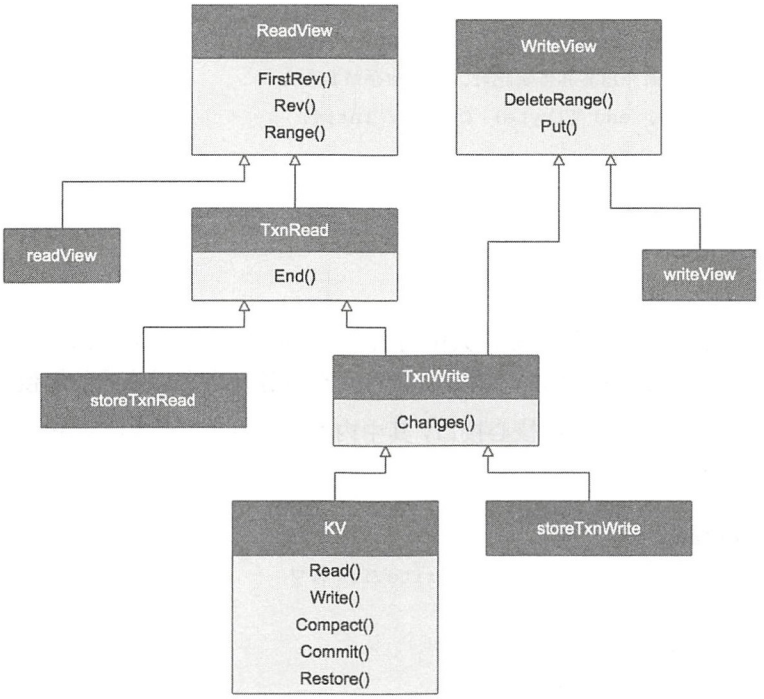


图 6-9

首先看一下 ReadView 接口，该接口定义了只读事务相关的视图方法，其定义如下所示。

```
type ReadView interface {
  // Rev()方法会返回开启当前只读事务时的 revision 信息
  Rev() int64

  // FirstRev()方法会返回开启当前只读事务时的 revision 信息，这与 Rev()方法相同，但是当进行一
  // 次压缩操作之后，该方法的返回值会被更新成压缩时的 revision 信息，也就是压缩后的最小 revision
  FirstRev() int64

  // 范围查询，在后面分析其具体实现时，再详细介绍其含义
  Range(key, end []byte, ro RangeOptions) (r *RangeResult, err error)
}
```

TxnRead 接口表示一个只读事务，其中内嵌了 ReadView 接口，并在其基础上扩展了一个 End()方法，该 End()方法用来表示当前事务已经完成，并准备提交。

WriteView 接口中定义了读写事务相关的方法，其定义如下所示。

```
type WriteView interface {
    // 范围删除，在后面介绍其具体实现时，再详细介绍其含义
    DeleteRange(key, end []byte) (n, rev int64)

    // 添加指定的键值对
    Put(key, value []byte, lease lease.LeaseID) (rev int64)
}
```

TxnWrite 接口表示一个读写事务，其中内嵌了 TxnRead 接口和 WriteView 接口，并在两者的基础上扩展了一个 Changes()方法，该方法会返回自事务开启之后修改的键值对信息。

KV 接口是最后一个要介绍的核心接口，其中内嵌了 ReadView 接口和 WriteView 接口，KV 接口的定义如下所示。

```
type KV interface {
    ReadView // 内嵌了 ReadView 接口和 WriteView 接口
    WriteView
    Read() TxnRead  // 创建只读事务
    Write() TxnWrite // 创建读写事务
    Compact(rev int64) (<-chan struct{}, error) // 对整个 KV 存储进行压缩
    Commit() // 提交事务
    Restore(b backend.Backend) error    // 从 BoltDB 中恢复内存索引
}
```

下面看一下上述接口的具体实现，首先是 readView 结构体，该结构体实现了 ReadView 接口。结构体 readView 中只有一个 kv 字段（KV 类型），结构体 readView 实现 FirstRev()方法、Rev()方法和 Range()方法的方式基本类似：先调用 kv.Read()获取只读事务（TxnRead 实例），然后调用 TxnRead 实例的对应方法完成相应操作，最后调用 TxnRead.End()方法结束事务。

结构体 writeView 实现了 WriteView 接口，其中也是只有一个 kv 字段（KV 类型）。writeView 实现 DeleteRange()、Put()的方式与上面介绍的 readView.FirstRev()方法等类似，这里不再展开描述了。

结构体 store（package mvcc）实现了 KV 接口，其中各个字段的含义如下所示。

- **mu（sync.RWMutex 类型）**：在开启只读/读写事务时，需要获取该读锁进行同步，即在 Read()方法和 Write()方法中获取该读锁，在 End()方法中释放。在进行压缩等非事务性的操作时，需要加写锁进行同步。
- **b（backend.Backend 类型）**：当前 store 实例关联的后端存储，Backend 接口及其具体

实现（backend 结构体）在前面已经详细介绍过了，这里不再赘述。

- kvindex（index 类型）：当前 store 实例关联的内存索引，index 接口及其具体实现（treeIndex）在前面的小节中已经详细介绍过了，这里不再赘述。
- currentRev（int64 类型）：该字段记录当前的 revision 信息（main revision 部分的值）。
- compactMainRev（int64 类型）：该字段记录最近一次压缩后最小的 revision 信息（main revision 部分的值）。
- revMu（sync.RWMutex 类型）：在修改 currentRev 字段和 compactMainRev 字段时，需要获取该锁进行同步。
- bytesBuf8（[]byte 类型）：索引缓冲区，主要用于记录 ConsistentIndex，在后面介绍 saveIndex()方法时，还会介绍该字段的作用。
- fifoSched（schedule.Scheduler 类型）：FIFO 调度器，在后面会详细介绍其具体实现。
- le（lease.Lessor 类型）：租约相关的内容，后面详细介绍 lease 相关的内容。

另外，store 还内嵌了 ReadView 接口和 WriteView 接口。下面先分析 NewStore()函数，其中完成了 store 实例的初始化过程，具体实现如下：

```
func NewStore(b backend.Backend, le lease.Lessor, ig ConsistentIndexGetter) *store {
  s := &store{
    b:       b, // 初始化 backend 字段
    kvindex: newTreeIndex(), // 初始化 kvindex 字段
    le: le,
    currentRev:     1, // currentRev 字段初始化为 1
    compactMainRev: -1, // compactMainRev 字段初始化为-1
    bytesBuf8: make([]byte, 8),
    fifoSched: schedule.NewFIFOScheduler(),
    stopc: make(chan struct{}),
  }
  // 创建 readView 实例和 writeView 实例
  s.ReadView = &readView{s}
  s.WriteView = &writeView{s}
  if s.le != nil {
    s.le.SetRangeDeleter(func() lease.TxnDelete { return s.Write() })
  }
  // 获取 backend 的读写事务，创建名为“key”和“meta”的两个 Bucket，然后提交事务
  tx := s.b.BatchTx()
  tx.Lock()
```

```
    tx.UnsafeCreateBucket(keyBucketName)
    tx.UnsafeCreateBucket(metaBucketName)
    tx.Unlock()
    s.b.ForceCommit()
    // 从 backend 中恢复当前 store 的所有状态，其中包括内存中的 BTree 索引等，错误处理(略)
    if err := s.restore();...
    return s
}
```

➢ Read

下面看一下 store 结构体实现的 Read()方法，在 store.Read()方法中会返回一个 storeTxnRead 实例（实现了 TxnRead 接口），其具体实现如下：

```
func (s *store) Read() TxnRead {
    s.mu.RLock() // 加读锁
    tx := s.b.ReadTx() // 获取只读事务
    s.revMu.RLock() // 要读取 store 中的 compactMainRev 字段和 currentRev 字段，加读锁同步
    tx.Lock()
    // 读取 store 中的 compactMainRev 字段和 currentRev 字段，用于创建后续的 storeTxnRead 实例
    firstRev, rev := s.compactMainRev, s.currentRev
    s.revMu.RUnlock() // 读取完 compactMainRev 字段和 currentRev 字段，释放读锁
    // metricsTxnWrite 同时实现了 TxnRead 和 TxnWrite 接口，并在原有的功能上记录监控信息
    return newMetricsTxnRead(&storeTxnRead{s, tx, firstRev, rev})
}
```

结构体 storeTxnRead 中的具体字段如下所示。

- s（*store 类型）：该 storeTxnrRead 实例关联的 store 实例。
- tx（backend.ReadTx 类型）：该 storeTxnrRead 实例关联的 ReadTx 实例。
- firstRev 和 rev（int64 类型）：对应于关联的 store 实例的 firstRev 和 rev 字段。

storeTxnRead 中实现的 FirstRev()和 Rev()方法比较简单，直接返回了 firstRev 字段和 rev 字段，这里不再展开介绍。

storeTxnRead.Range()方法是通过调用其 rangeKeys()方法实现的。在开始介绍 storeTxnRead.rangeKeys()方法的具体实现之前，先来了解其参数和返回值。首先是结构体 RangeOptions，它是 rangeKeys()方法的参数之一，其中封装的信息如下所示。

- Limit（int64 类型）：此次查询返回的键值对个数的上限。

- Rev（int64 类型）：扫描内存索引时使用到的 main revision 部分的值。
- Count（bool 类型）：如果将该值设置为 true，则只返回键值对个数，并不返回具体的键值对数据。

storeTxnRead.rangeKeys()方法的返回值类型是 RangeResult，其中封装的核心信息如下所示。

- KVs（[]mvccpb.KeyValue 类型）：此次查询得到的键值对数据集合。
- Count（int 类型）：此次查询返回的键值对个数。

最后来看一下 KeyValue 结构体，从名字就能看出，KeyValue 中封装了原始的键值对数据，其中各个字段的含义如下所示。

- Key（[]byte 类型）：原始的 Key 值。
- CreateRevision（int64 类型）：最近一次创建当前键值对时的 main revision 值。
- ModRevision（int64 类型）：最近一次修改当前键值对时的 main revision 值。
- Version（int64 类型）：当前键值对的版本，每当修改当前键值对时，都会使得该字段递增。当删除当前键值对时，该值会重置为 0。
- Value（[]byte 类型）：原始的 Value 值。
- Lease（int64）：关联的 Lease 实例 ID，当关联的 Lease 实例过期，则当前键值对也会被删除。如果该字段为 0，则表示没有 Lease 实例与当前键值对关联。关于 Lease 的内容，后面会详细介绍。

了解了具体的参数和返回值之后，我们再看一下 storeTxnRead.rangeKeys()方法的具体实现。该方法的大致流程是：首先扫描内存索引（即前面介绍的 treeIndex），得到对应的 revision，然后通过 revision 查询 BoltDB 得到真正的键值对数据，最后将键值对数据反序列化成 KeyValue 并封装成 RangeResult 实例返回。storeTxnRead.rangeKeys()方法的具体实现如下所示。

```
func (tr *storeTxnRead) rangeKeys(key, end []byte, curRev int64, ro RangeOptions)
      (*RangeResult, error) {
  rev := ro.Rev
  // 检测 RangeOptions 中指定的 revision 信息是否合法，如果不合法，则根据 curRev（对 storeTxnRead
  // 来说，就是 rev 字段）当前的 compactMainRev 值对 rev 进行校正（略）
  // 调用 index.Range()方法查询指定范围的键值对信息
  _, revpairs := tr.s.kvindex.Range(key, end, int64(rev))
  // 根据 RangeOptions 中指定的参数，创建 RangeResult 实例作为返回值
  if len(revpairs) == 0 {
    return &RangeResult{KVs: nil, Count: 0, Rev: curRev}, nil
  }
```

仅供非商业用途或交流学习使用


```
    if ro.Count { // 只获取键值对的个数，不需要返回具体的键值对数据，所以只查询索引即可
        return &RangeResult{KVs: nil, Count: len(revpairs), Rev: curRev}, nil
    }

    var kvs []mvccpb.KeyValue
    for _, revpair := range revpairs { // 遍历查询到的 revision 信息
        start, end := revBytesRange(revpair) // 将 revision 拆分成两个 key
        // 根据 revision 实例拆分后的结果，对 BoltDB 进行查询
        _, vs := tr.tx.UnsafeRange(keyBucketName, start, end, 0)
        if len(vs) != 1 { // 虽然使用了 UnsafeRange()方法进行查询，但是其查询结果必然只有一个
            plog.Fatalf("range cannot find rev (%d,%d)", revpair.main, revpair.sub)
        }
        //将查询得到的键值对数据进行反序列化，得到 KeyValue 实例，并追加到 kvs 中
        var kv mvccpb.KeyValue
        if err := kv.Unmarshal(vs[0]);...
        kvs = append(kvs, kv) //
        if ro.Limit > 0 && len(kvs) >= int(ro.Limit) {
            break
        }
    }
    // 将 kvs 封装成 RangeResult 实例并返回
    return &RangeResult{KVs: kvs, Count: len(revpairs), Rev: curRev}, nil
}
```

这里简单说明 revBytesRange()函数是如何将一个 revision 实例转换成两个 key 的，具体实现如下：

```
func revBytesRange(rev revision) (start, end []byte) {
    start = newRevBytes() // 新建一个[]byte 切片
    // 将 main revision 部分和 sub revision 部分写入 start 中，两者通过下画线分隔
    revToBytes(rev, start)
    end = newRevBytes() // 创建 end 的过程，与 start 同理，只不过 sub revision 增加 1
    endRev := revision{main: rev.main, sub: rev.sub + 1}
    revToBytes(endRev, end)
    return start, end
}
```

在 storeTxnRead.End()方法中，会释放在 store.Read()方法中获取的 store.mu 上的读锁和底层

仅供非商业用途或交流学习使用

只读事务上的锁，前面已经介绍过 readTx.Unlock()方法的具体实现，这里不再展开详细介绍。

➢ Write

介绍完 store.Read()方法相关的内容之后，再来分析 store.Write()方法的实现。store.Write()方法会返回 storeTxnWrite 实例，其具体实现如下所示。

```
func (s *store) Write() TxnWrite {
  s.mu.RLock()// 加读锁
  tx := s.b.BatchTx() // 获取读写事务
  tx.Lock() // 获取读写事务的锁
  tw := &storeTxnWrite{ // 创建 storeTxnWrite 实例
    storeTxnRead: &storeTxnRead{s, tx, 0, 0}, // 其中 first    Rev 初始化为 0
    tx:          tx,
    beginRev:     s.currentRev,
    changes:      make([]mvccpb.KeyValue, 0, 4),
  }
  // metricsTxnWrite 的主要功能与前面介绍的 metricsTxnRead 类似，这里不再赘述
  return newMetricsTxnWrite(tw)
}
```

结构体 storeTxnWrite 实现了 TxnWrite 接口，其中内嵌了 storeTxnRead，其他字段的含义如下所示。

- tx（backend.BatchTx 类型）：当前 storeTxnWrite 实例关联的读写事务。
- beginRev（int64 类型）：记录创建当前 storeTxnWrite 实例时 store.currentRev 字段的值。
- changes（[]mvccpb.KeyValue 类型）：在当前读写事务中发生改动的键值对信息。

下面先看一下 storeTxnWrite.Put()方法的实现，该方法实现了向存储中追加一个键值对数据的功能，其底层是通过调用 storeTxnWrite.put()方法实现的，具体实现如下所示。

```
func (tw *storeTxnWrite) put(key, value []byte, leaseID lease.LeaseID) {
  rev := tw.beginRev + 1 // 当前事务产生的修改对应的 main revision 部分都是该值
  c := rev
  // 在内存索引中查找对应的键值对信息，在后面创建 KeyValue 实例时会使用到这些值
  _, created, ver, err := tw.s.kvindex.Get(key, rev)
  if err == nil {
    c = created.main
  }
```

仅供非商业用途或交流学习使用


```
    ibytes := newRevBytes()
    // 创建此次 Put 操作对应的 revision 实例，其中 main revision 部分为 beginRev + 1
    idxRev := revision{main: rev, sub: int64(len(tw.changes))}
    // 将 main revision 和 sub revision 两部分写入 ibytes 中，后续会作为写入 BoltDB 的 Key
    revToBytes(idxRev, ibytes)

    ver = ver + 1
    kv := mvccpb.KeyValue{ // 创建 KeyValue 实例
      Key:            key, // 原始 Key 值
      Value:          value, // 原始 Value 值
      // 如果内存索引中已经存在该键值对，则创建版本不变，否则为新建 Key，则将 CreateRevision 设
      // 置成 beginRev + 1
      CreateRevision: c,
      ModRevision:    rev, //  beginRev + 1
      Version:        ver, // 递增 verison
      Lease:          int64(leaseID),
    }
    d, err := kv.Marshal() // 将上面创建的 KeyValue 实例序列化，错误处理(略)

    // 将 Key(main revision 和 sub revision 组成)和 Value(上述 KeyValue 实例序列化的结果)写入
    // BoltDB 中名为“key”的 Bucket
    tw.tx.UnsafeSeqPut(keyBucketName, ibytes, d)
    tw.s.kvindex.Put(key, idxRev) // 将原始 Key 与 revision 实例的对应关系写入内存索引中
    tw.changes = append(tw.changes, kv) // 将上述 KeyValue 写入到 changes 中
    // 下面关于 Lease 的逻辑暂时省略，后面会有专门的章节介绍 Lease 的内容
}
```

storeTxnWrite.Range()方法也是调用其 rangeKeys()方法实现的，与 storeTxnRead.Range()方法的主要区别在于传入的 rev 参数。storeTxnWrite.Range()方法的具体实现如下：

```
func (tw *storeTxnWrite) Range(key, end []byte, ro RangeOptions)
     (r *RangeResult, err error) {
  rev := tw.beginRev
  if len(tw.changes) > 0 {
    // 如果当前读写事务中有更新键值对的操作，则该方法也需要能查询到这些更新的键值对，
    // 所以传入的 rev 参数是 beginRev+1
    rev++
  }
```

仅供非商业用途或交流学习使用


```
    return tw.rangeKeys(key, end, rev, ro) // rangeKeys()方法在前面介绍过了，这里不再赘述
}
```

storeTxnWrite.DeleteRange()方法是通过调用 deleteRange()方法实现的，如下所示。

```
func (tw *storeTxnWrite) DeleteRange(key, end []byte) (int64, int64) {
    if n := tw.deleteRange(key, end); n != 0 || len(tw.changes) > 0 {
        return n, int64(tw.beginRev + 1) // 根据当前事务是否有更新操作，决定返回的main revision值
    }
    return 0, int64(tw.beginRev)
}
```

在 storeTxnWrite.deleteRange()方法中，首先会从内存索引查询待删除的 Key 和对应的 revision 实例，然后会调用 storeTxnWrite.delete()方法逐个删除，其具体实现如下：

```
func (tw *storeTxnWrite) deleteRange(key, end []byte) int64 {
    rrev := tw.beginRev
    if len(tw.changes) > 0 {  // 根据当前事务是否有更新操作，决定后续使用的main revision值
        rrev += 1
    }
    // 在内存索引中查询待删除的key
    keys, revs := tw.s.kvindex.Range(key, end, rrev)
    if len(keys) == 0 {
        return 0
    }
    for i, key := range keys { // 遍历待删除的key，并调用delete()方法完成删除
        tw.delete(key, revs[i])
    }
    return int64(len(keys)) // 返回删除的key值
}
```

在 storeTxnWrite.delete()方法中会向 BoltDB 和内存索引中添加 tombstone 键值对，两者对 tombstone 键值对的处理在前面已经详细分析过了，这里不再赘述。这里重点来看 storeTxnWrite.delete()方法的具体实现：

```
func (tw *storeTxnWrite) delete(key []byte, rev revision) {
    // 将revision转换成BoltDB中的key
    ibytes := newRevBytes()
```

仅供非商业用途或交流学习使用


```
    idxRev := revision{main: tw.beginRev + 1, sub: int64(len(tw.changes))}
    revToBytes(idxRev, ibytes)
    // 追加一个't'标识 Tombstone
    ibytes = appendMarkTombstone(ibytes)
    // 创建 KeyValue 实例，其中只包含 key，并进行序列化
    kv := mvccpb.KeyValue{Key: key}
    d, err := kv.Marshal()
    // 将上面生成的 key(ibytes)和 value(KeyValue 序列化后的数据)写入名为“key”的 Bucket 中
    tw.tx.UnsafeSeqPut(keyBucketName, ibytes, d)
    // 在内存索引中添加 Tombstone，错误处理(略)
    err = tw.s.kvindex.Tombstone(key, idxRev)
    tw.changes = append(tw.changes, kv) // 向 changes 中追加上述 KeyValue 实例
    // 下面关于 Lease 的逻辑暂时省略，后面会有专门的章节介绍 Lease 的内容
}
```

storeTxnWrite.Changes()方法直接返回其 changes 字段，实现比较简单，这里不再赘述。在 storeTxnWrite 中最后一个要介绍的是 End()方法，其中主要完成三个操作，一是递增 store.currentRev，二是将 ConsistentIndex 记录到名为“meta”的 Bucket 中，三是调用 Unlock()方法提交当前事务并开启新的读写事务。storeTxnWrite.End()方法的具体实现如下：

```
func (tw *storeTxnWrite) End() {
    if len(tw.changes) != 0 { // 检测当前读写事务中是否有修改操作
        // 将 ConsistentIndex(实际就是当前处理的最后一条 Entry 记录的索引值)添加记录到 BoltDB
        // 中名为“meta”的 Bucket 中
        tw.s.saveIndex(tw.tx)
        tw.s.revMu.Lock() // 修改 store 中的 currentRev 字段，加读锁同步
        tw.s.currentRev++ // 递增 currentRev
    }
    // 在前面介绍过 batchTxBuffered 及 batchTx 中的 Unlock()方法实现，其中会将当前读写事务提交，
    // 同时开启新的读写事务
    tw.tx.Unlock()
    if len(tw.changes) != 0 {
        tw.s.revMu.Unlock() // 修改 currentRev 完成，释放读锁
    }
    tw.s.mu.RUnlock() // 释放读锁
}

// saveIndex()方法的具体实现如下所示
```

仅供非商业用途或交流学习使用


```
func (s *store) saveIndex(tx backend.BatchTx) {
    bs := s.bytesBuf8
    // 将 ConsistentIndex 值写入 bytesBuf8 缓冲区中
    binary.BigEndian.PutUint64(bs, s.ig.ConsistentIndex())
    // 将 ConsistentIdnex 值记录到 meta Bucket 中
    tx.UnsafePut(metaBucketName, consistentIndexKeyName, bs)
}
```

➢ Compact

从该方法的名字就可以看出，store.Compact()方法主要用于实现键值对的压缩。该方法首先将传入的参数转换成 revision 实例，然后在 meta Bucket 中记录此次压缩的键值对信息（Key 为 scheduledCompactRev，Value 为 revision），之后调用 treeIndex.Compact()方法完成对内存索引的压缩，最后通过 FIFO Scheduler 异步完成对 BoltDB 的压缩操作。下面紧接着分析 store.Compact()方法的具体实现：

```
func (s *store) Compact(rev int64) (<-chan struct{}, error) {
    // 获取 mu 上的写锁和 revMu 上的写锁(略)
    // 检测传入的 rev 参数是否合法(略)
    start := time.Now() // 记录当前时间，主要是为了后面记录监控，监控的相关代码(略)
    s.compactMainRev = rev // 更新 compactMainRev
    // 将传入的 rev 值封装成 revision 实例，并写入[]byte 切片中
    rbytes := newRevBytes()
    revToBytes(revision{main: rev}, rbytes)
    // 获取 BoltDB 中的读写事务，并将上面创建的 revision 实例写入到名为“meta”的 Bucket 中
    tx := s.b.BatchTx()
    tx.Lock()
    tx.UnsafePut(metaBucketName, scheduledCompactKeyName, rbytes)
    tx.Unlock()
    s.b.ForceCommit() // 提交当前读写事务

    // 对内存索引进行压缩，其具体实现在前面已经分析过了，这里不再赘述
    // 其返回值是此次压缩过程中涉及的 revision 实例
    keep := s.kvindex.Compact(rev)
    ch := make(chan struct{})
    var j = func(ctx context.Context) {
        // 检测 ctx(略)
        // 这里是真正对 BoltDB 进行压缩的地方。省略压缩出现失败的场景处理
```

仅供非商业用途或交流学习使用


```
        if !s.scheduleCompaction(rev, keep) ...
        close(ch) // 当正常压缩介绍之后，会关闭该通道，通知其他监听的
    }

    // store.fifoSched 是一个 FIFO Scheduler，在 Scheduler 接口中定义了一个 Schedule(j Job)
    // 方法，该方法会将 Job 放入调度器进行调度，其中的 Job 实际上就是一个 func(context.Context) 函数
    // 这里将对 BoltDB 的压缩操作（即上面定义的 j 函数）放入 FIFO 调度器中，异步执行
    s.fifoSched.Schedule(j)
    return ch, nil
}
```

store.scheduleCompaction()方法中完成了对 BoltDB 中储存的键值对的压缩，在该方法会通过 UnsafeRange()方法批量查询待删除的 Key（revision），然后逐个调用 UnsafeDelete()方法进行删除，全部的待压缩 Key 被处理完成之后，会向 mete Bucket 中写入此次压缩的相关信息，其中 Key 为“finishedCompactRev”。scheduleCompaction()方法的具体实现如下：

```
func (s *store) scheduleCompaction(compactMainRev int64, keep map[revision]struct{})
bool {
    last := make([]byte, 8+1+8) // 范围查询的起始 key(revision)
    // 这里将compactMainRev+1写入end([]byte类型)中(Compact()方法中已经更新过compactMainRev)，
    // 作为范围查询的结束 key(revision)
    end := make([]byte, 8)
    binary.BigEndian.PutUint64(end, uint64(compactMainRev+1))
    batchsize := int64(10000) // 范围查询的上限
    for {
        var rev revision
        tx := s.b.BatchTx() // 获取读写事务
        tx.Lock() // 加锁
        // 调用 UnsafeRange()方法对 key Bucket 进行范围查询
        keys, _ := tx.UnsafeRange(keyBucketName, last, end, batchsize)
        // 遍历上述查询到的 Key(revision)，并逐个调用 UnsafeDelete()方法进行删除
        for _, key := range keys {
            rev = bytesToRev(key)
            if _, ok := keep[rev]; !ok {
                tx.UnsafeDelete(keyBucketName, key)
            }
        }
```

仅供非商业用途或交流学习使用


```
        if len(keys) < int(batchsize) { // 最后一次批量处理
            // 将compactMainRev封装成revision,并写入meta Bucket中,其Key为"finishedCompactRev"
            rbytes := make([]byte, 8+1+8)
            revToBytes(revision{main: compactMainRev}, rbytes)
            tx.UnsafePut(metaBucketName, finishedCompactKeyName, rbytes)
            tx.Unlock() // 读写事务使用完毕, 解锁
            return true
        }

        // 更新last, 下次范围查询的起始key依然是last
        revToBytes(revision{main: rev.main, sub: rev.sub + 1}, last)
        tx.Unlock()
        dbCompactionPauseDurations.Observe(float64(time.Since(start) / time.Millisecond))

        select { // 每次范围处理结束之后
        case <-time.After(100 * time.Millisecond):
        case <-s.stopc:
            return false
        }
    }
}
```

➢ Restore

在介绍 Raft 协议时提到，Follower 节点在收到快照数据时，会使用快照数据恢复当前节点的状态。在这个恢复过程中就会调用 store.Restore()方法完成内存索引和 store 中其他状态的恢复，其具体实现如下：

```
func (s *store) Restore(b backend.Backend) error {
    // 获取mu上的写锁, 关闭当前的FIFO Scheduler(略)
    s.b = b // 更新store.b字段, 指向新的backend实例
    s.kvindex = newTreeIndex() // 更新store.kvindex字段, 指向新的内存索引
    s.currentRev = 1 // 重置currentRev字段和compactMainRev字段
    s.compactMainRev = -1
    s.fifoSched = schedule.NewFIFOScheduler()      // 更新fifoSched字段, 指向新的FIFO
                                                   // Scheduler
    return s.restore() // 调用restore()开始的恢复内存索引
}
```

仅供非商业用途或交流学习使用

store.restore()方法恢复内存索引的大致逻辑是：首先批量读取 BoltDB 中所有的键值对数据，然后将每个键值对封装成 revKeyValue 实例，并写入一个通道中，最后由另一个单独的 goroutine 读取该通道，并完成内存索引的恢复。过程如图 6-10 所示。

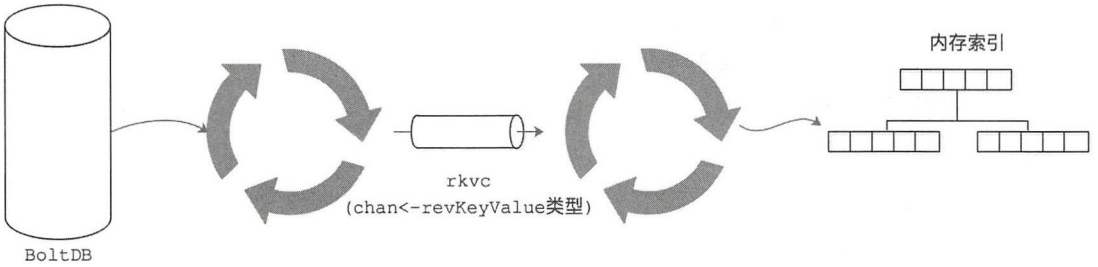


图 6-10

在开始分析 store.restore()方法之前，先来了解一下结构体 revKeyValue，其中字段的含义如下所示。

- key（[]byte 类型）：BoltDB 中的 Key 值，可以转换得到的 revision 实例。
- kv（mvccpb.KeyValue 类型）：BoltDB 中保存的 Value 值。
- kstr（string 类型）：原始的 Key 值。

store.restore()方法的具体实现如下：

```
func (s *store) restore() error {
  b := s.b
  // 创建 min 和 max，后续在 BoltDB 中进行范围查询时的起始 key 和结束 key 就是 min 和 max
  min, max := newRevBytes(), newRevBytes()
  revToBytes(revision{main: 1}, min)   // min 为 1_0
  // max 为 MaxInt64_MaxInt64
  revToBytes(revision{main: math.MaxInt64, sub: math.MaxInt64}, max)

  tx := s.b.BatchTx() // 获取读写事务，并加锁
  tx.Lock()
  // 调用 UnsafeRange()方法，在 meta Bucket 中查询上次的压缩完成时的相关
  // 记录(Key 为 finishedCompactRev)
  _, finishedCompactBytes := tx.UnsafeRange(metaBucketName, finishedCompactKeyName,
      nil, 0)
  if len(finishedCompactBytes) != 0 { // 根据查询结果，恢复 store.compactMainRev 字段
    s.compactMainRev = bytesToRev(finishedCompactBytes[0]).main
  }
```

仅供非商业用途或交流学习使用


```
// 调用 UnsafeRange()方法，在 meta Bucket 中查询上次的压缩启动时的
// 相关记录(Key 为 scheduledCompactRev)
_, scheduledCompactBytes := tx.UnsafeRange(metaBucketName, scheduledCompactKeyName,
    nil, 0)
scheduledCompact := int64(0)
if len(scheduledCompactBytes) != 0 {// 根据查询结果，更新 scheduledCompact 变量
  scheduledCompact = bytesToRev(scheduledCompactBytes[0]).main
}

// 在 restoreIntoIndex()方法中会启动一个单独的 goroutine，用于接收从 backend 中读取的键值
// 对数据，并恢复到新建的内存索引中(store.kvindex)，稍后会详细分析该方法的具体实现
// 这里需要了解该方法的返回值 rkvc 是一个通道，也是当前 goroutine 与上述 goroutine 通信的桥梁
rkvc, revc := restoreIntoIndex(s.kvindex)
for {
  // 调用 UnsafeRange()方法查询 BoltDB 中的 key Bucket，返回键值对数量的上限是 restoreChunkKeys，
  // 默认是 10000
  keys, vals := tx.UnsafeRange(keyBucketName, min, max, int64(restoreChunkKeys))
  if len(keys) == 0 {  // 查询结果为空，则直接结束当前 for 循环
    break
  }
  // 将查询到的键值对数据写入 rkvc 这个通道中，并由 restoreIntoIndex()方法中创建的
  // goroutine 进行处理，稍后会详细介绍 restoreChunk()方法的具体实现
  restoreChunk(rkvc, keys, vals, keyToLease)
  if len(keys) < restoreChunkKeys {
    break // 范围查询得到的结果数小于 restoreChunkKeys，即表示最后一次查询
  }
  // 更新 min 作为下一次范围查询的起始 key
  newMin := bytesToRev(keys[len(keys)-1][:revBytesLen])
  newMin.sub++
  revToBytes(newMin, min)
}
close(rkvc) // 关闭 rkvc 通道
s.currentRev = <-revc // 从 revc 通道中获取恢复之后的 currentRev
// 校正 currentRev 和 scheduledCompact(略)
// 租期相关的处理，后面详细介绍(略)
tx.Unlock() // 读写事务结束，释放锁
if scheduledCompact != 0 { // 如果在开始恢复之前，存在未执行完的压缩操作，则重启该压缩操作
  s.Compact(scheduledCompact)
```

仅供非商业用途或交流学习使用


```
    }
    return nil
}
```

在 store.restore()方法中，通过 UnsafeRange()方法从 BoltDB 中查询到的键值对，然后通过 restoreChunk()函数转换成 revKeyValue 实例，并写入 rkvc 通道中。store.restore()方法的具体实现如下所示。

```
func restoreChunk(kvc chan<- revKeyValue, keys, vals [][]byte,
      keyToLease map[string]lease.LeaseID) {
  for i, key := range keys { // 遍历读取到的 revision 信息
    rkv := revKeyValue{key: key} // 创建 revKeyValue 实例
    if err := rkv.kv.Unmarshal(vals[i]); ... // 反序列化得到 KeyValue，错误处理(略)
    rkv.kstr = string(rkv.kv.Key) // 记录对应的原始 Key 值
    if isTombstone(key) {
      delete(keyToLease, rkv.kstr) // 删除 tombstone 标识的键值对
    }
    // 省略 Lease 相关的处理，后面详细介绍
    kvc <- rkv // 将上述 revKeyValue 实例写入 rkvc 通道中，等待处理
  }
}
```

启动一个后台的 goroutine，读取 rkvc 通道中的 revKeyValue 实例，并将其中的键值对数据恢复到内存索引中等一系列操作，都是在 restoreIntoIndex ()函数中完成的，其具体实现如下所示。

```
func restoreIntoIndex(idx index) (chan<- revKeyValue, <-chan int64) {
  // 创建两个通道，其中 rkvc 通道就是前面一直提到的，用来传递 revKeyValue 实例的通道
  rkvc, revc := make(chan revKeyValue, restoreChunkKeys), make(chan int64, 1)
  go func() { //  启动一个单独的 goroutine
    currentRev := int64(1) // 记录当前遇到的最大的 main revision 值
    // 在该 goroutine 结束时（即 rkvc 通道被关闭时），将当前已知的最大 main revision 值写入
    // revc 通道中，待其他 goroutine 读取
    defer func() { revc <- currentRev }()
    // 虽然 BTree 的查找效率很高，但是随着 BTree 层次的加深，效率也随之下降，这里使用 kiCache 这
    // 个 map 做了一层缓存，更加高效地查找对应的 keyIndex 实例
    // 前面从 BoltDB 中读取到的键值对数据并不是按照原始 Key 值进行排序的，如果直接向 BTree 中写入，
    // 则可能会引起节点的分裂等变换操作，效率比较低。所以这里使用 kiCache 这个 map 做了一层缓存
    kiCache := make(map[string]*keyIndex, restoreChunkKeys)
```

仅供非商业用途或交流学习使用


```
    for rkv := range rkvc { // 从 rkvc 通道中读取 revKeyValue 实例
        ki, ok := kiCache[rkv.kstr] // 先查询缓存
        // 如果 kiCache 中缓存了大量的 Key，但是依然没有命中，则清理缓存
        if !ok && len(kiCache) >= restoreChunkKeys {
            i := 10
            for k := range kiCache {
                delete(kiCache, k)
                if i--; i == 0 { // 只清理 10 个 Key
                    break
                }
            }
        }
        if !ok { // 如果缓存未命中，则从内存索引中查找对应的 keyIndex 实例，并添加到缓存中
            ki = &keyIndex{key: rkv.kv.Key}
            if idxKey := idx.KeyIndex(ki); idxKey != nil {
                kiCache[rkv.kstr], ki = idxKey, idxKey
                ok = true
            }
        }
        rev := bytesToRev(rkv.key) // 将 revKeyValue.key 转换成 revision 实例
        currentRev = rev.main // 更新 currentRev 值
        if ok {
            if isTombstone(rkv.key) { // 当前 revKeyValue 实例对应一个 tombstone 键值对
                ki.tombstone(rev.main, rev.sub) // 在对应的 keyIndex 实例中插入 tombstone
                continue
            }
            ki.put(rev.main, rev.sub) // 在对应 keyIndex 实例中添加正常的 revision 信息
        } else if !isTombstone(rkv.key) {
            // 如果从内存索引中依然未查询到对应的 keyIndex 实例，则需要填充 keyIndex 实例中其他字段，
            // 并添加到内存索引中
            ki.restore(revision{rkv.kv.CreateRevision, 0}, rev, rkv.kv.Version)
            idx.Insert(ki)
            kiCache[rkv.kstr] = ki // 同时会将该 keyIndex 实例添加到 kiCache 缓存中
        }
    }
}()
return rkvc, revc
}
```

仅供非商业用途或交流学习使用

既然结构体 store 实现了 KV 接口，当然也实现了 KV.Commit()方法，该方法除了会提交事务，还会调用前面介绍的 saveIndex()方法记录 ConsistentIndex，其实现比较简单，留给读者自己进行分析。到此为止，store 中比较核心的方法就全部介绍完了。v3 版本存储中关于后端存储和内存索引的原理及实现也全部分析完了。

6.2.2 watcher机制

在上一节中，我们已经完整地分析了 v3 存储中的内存索引和后端存储的原理及具体实现。在本节中，我们重点介绍 v3 版本存储中的 watcher 机制的相关实现。

在 v3 版本存储中，watcher 机制支持监听某个固定的 Key 值，也支持监听一个范围，这有点类似于 v2 版本存储中监听一个目录节点及其子节点的效果（读者可以回顾前面关于 v2 版本的后端存储的相关内容）。

1. watcher、Event、WatchResponse

我们先来看一下 watcher 结构体，它是整个 Watcher 机制的基础，其中各个字段的含义如下所示。

- id（WatchID 类型）：当前 watcher 实例的唯一标识。
- key（[]byte 类型）：该 watcher 实例监听的原始 Key 值。
- end（[]byte 类型）：该 watcher 实例监听的结束位置（也是一个原始 Key 值）。如果该字段有值，则当前 watcher 实例是一个范围 watcher。如果该字段未设置值，则当前 watcher 只监听上面的 key 字段对应的键值对。
- ch（chan<– WatchResponse 类型）：当前 watcher 实例被触发之后，会向该通道中写入 WatchResponse。该通道可能是由多个 watcher 实例共享的。
- victim（bool 类型）：当上述 ch 通道阻塞时，会将该字段设置成 true。
- compacted（bool 类型）：如果该字段被设置为 true，则表示当前 watcher 已经因为发生了压缩操作而被删除。
- minRev（int64 类型）：能够触发当前 watcher 实例的最小 revision 值。发生在该 revision 之前的更新操作是无法触发该 watcher 实例。
- fcs（[]FilterFunc 类型）：过滤器。触发当前 wathcer 实例的事件（后面介绍的 Event 结构体就是对“事件”的抽象）需要经过这些过滤器的过滤才能封装进响应（后面介绍的 WatchResponse 就是对“响应”的抽象）中。

当 watcher 实例监听的 Key 发生了变化，则使用一个 Event 实例进行表示。结构体 Event

仅供非商业用途或交流学习使用

中各个字段的含义如下。

- Type（int32 类型）：如果该字段为 0，则该 Event 实例对应一个 PUT 事件，即对应的键值对发生了新增或更新；如果该字段为 1，则该 Event 实例对应一个 DELETE 事件，即对应的键值对被删除。
- Kv（*KeyValue 类型）：该字段记录发生此次事件之后的键值对数据。
- PrevKv（*KeyValue 类型）：该字段记录发生此次事件之前的键值对数据。

为了提高效率，etcd 的服务端一般会批量处理 watcher 事件，在结构体 eventBatch 中可以封装多个 Event 实例，其中各个字段的含义如下。

- evs（[]mvccpb.Event 类型）：其中记录的 Event 实例是按照 revision 排序的。
- revs（int 类型）：记录当前 eventBatch 中记录的 Event 实例来自多少个不同的 main revision，特别注意，是不相同的 main revision 值的个数。
- moreRev（int64 类型）：由于当前 eventBatch 中记录的 Event 个数达到上限之后，后续 Event 实例无法加入该 eventBatch 中，该字段记录了无法加入该 eventBatch 实例的第一个 Event 实例对应的 main revision 值。

eventBatch.add()方法实现了将 Event 实例添加到 eventBatch 中保存的功能，其具体实现如下：

```
func (eb *eventBatch) add(ev mvccpb.Event) {
  // 检测 revs 是否达到上限，如果达到上限，则直接返回(略)
  if len(eb.evs) == 0 { // 第一次添加 Event 实例
    eb.revs = 1
    eb.evs = append(eb.evs, ev) // 将 Event 实例添加到 evs 字段中保存
    return
  }
  // 获取最后一个 Event 实例对应的 main revision 值
  ebRev := eb.evs[len(eb.evs)-1].Kv.ModRevision
  // 新增 Event 实例对应的 main revision 值
  evRev := ev.Kv.ModRevision
  if evRev > ebRev { // 比较两个 main revision 值
    eb.revs++ // 递增 revs 字段
    if eb.revs > watchBatchMaxRevs {
      // 当前 eventBatch 实例中记录的 Event 实例已达上限，则更新 moreRev 字段，
      // 记录最后一个无法加入的 Event 实例的 main revision 值
      eb.moreRev = evRev
      return
```

仅供非商业用途或交流学习使用


```
        }
    }
    eb.evs = append(eb.evs, ev) // 如果能继续添加 Event 实例，则将 Event 实例添加到 evs 中
}
```

WatchResponse 表示一次响应，其中封装了多个 Event 实例，watchResponse 中各个字段的含义如下。

- WatchID（WatchID 类型）：被触发的 watcher 实例的唯一标识。
- Events（[]mvccpb.Event 类型）：触发对应 watcher 事件集合。
- Revision（int64 类型）：当前 watchResponse 实例创建时对应的 revision 值，后面会详细介绍该字段的取值。
- CompactRevision（int64 类型）：如果因为压缩操作对应 watcher 实例被取消了，则该字段设置为压缩操作对应的 revision。

2. watchableStore

介绍完 watcher 机制中使用到的结构体之后，我们来分析 watchableStore 的具体实现。watchableStore 结构体完成了注册 watcher 实例、管理 watcher 实例，以及发送触发 watcher 之后的响应等核心功能。在结构体 watchableStore 中内嵌了 store，其它字段的含义如下所示。

- synced 和 unsynced（watcherGroup 类型）：在结构体 watchableStore 中包含两个 watcherGroup 实例，即 synced 字段和 unsynced 字段。synced watcherGroup 中全部的 watcher 实例都已经同步完毕，并等待新的更新操作；synced watcherGroup 中的 watcher 实例都落后于当前最新更新操作，并且有一个单独的后台 goroutine 帮助其进行追赶。当 etcd 服务端收到客户端的 watch 请求时，如果请求携带了 revision 参数，则比较该请求的 revision 信息和 store.currentRev 信息：如果请求中的 revision 信息较大，则放入 synced watcherGroup 中，否则放入 unsynced watcherGroup。在后面的分析中会看到，watchableStore 实例会启动一个后台的 goroutine 持续同步 unsynced watcherGroup，然后将完成同步的 watcher 实例迁移到 synced watcherGroup 中储存。
- victims（[]watcherBatch 类型）：如果 watcher 实例关联的 ch 通道被阻塞了，则对应的 watcherBatch 实例会暂时记录到该字段中。在 watchableStore 实例中还会启动一个后台 goroutine 来处理该字段中保存的 watcherBatch 实例。watcherBatch 实际上就是 map[*watcher]*eventBatch 类型，watcherBatch 只提供了一个 add()方法，用来添加被触发的 watcher 实例与其 eventBatch 之间的映射关系。
- victimc（chan struct{}类型）：当有新的 watcherBatch 实例添加到 victims 字段中时，会向该通道中发送一个空结构体作为信号。

仅供非商业用途或交流学习使用

- mu（sync.RWMutex 类型）：在修改 synced watcherGroup、unsynced watcherGroup 等字段时，需要获取该锁进行同步。
- wg（sync.WaitGroup 类型）：在 watchableStore 实例中会启动两个后台 goroutine，在 watchableStore.Close()方法中会通过该 sync.WaitGroup 实例实现等待两个后台 goroutine 执行完成的功能。

接下来分析 newWatchableStore()函数的具体实现。newWatchableStore()函数完成了 watchableStore 实例的初始化，其中会启动上面提到的两个后台 goroutine，其具体实现如下：

```
func newWatchableStore(b backend.Backend, le lease.Lessor, ig ConsistentIndexGetter)
        *watchableStore {
  s := &watchableStore{
    store:    NewStore(b, le, ig), // 创建 store 实例
    victimc:  make(chan struct{}, 1),
    unsynced: newWatcherGroup(), // 初始化 unsynced watcherGroup
    synced:   newWatcherGroup(), // 初始化 synced watcherGroup
    stopc:    make(chan struct{}),
  }
  // 创建 readView 和 writeView 实例
  s.store.ReadView = &readView{s}
  s.store.WriteView = &writeView{s}
  s.wg.Add(2) // 初始化 sync.WaitGroup 实例
  go s.syncWatchersLoop() // 启动处理 unsynced watcherGroup 的后台 goroutine
  go s.syncVictimsLoop()  // 启动处理 victims 的后台 goroutine
  return s
}
```

在开始介绍 watchableStore 结构体提供的方法之前，我们需要先了解一下 watcherGroup 结构体，其各个字段的含义如下所示。

- watchers（watcherSet 类型）：该字段中记录了当前 watcherGroup 实例中全部的 watcher 实例，这里的 watcherSet 实际上就是 map[*watcher]struct{}类型。在 watcherSet 中定义了 add()、delete()和 union()三个方法，分别用来添加、删除 watcher 实例和合并其他 watcherSet 实例，这些方法的实现并不复杂，这里不再展开分析，感兴趣的读者可以查看源码进行学习。
- keyWatchers（watcherSetByKey 类型）：该字段记录监听单个 Key 的 watcher 实例，watcherSetByKey 实际上就是 map[string]watcherSet 类型，该 map 中的 key 就是监听的原始 Key 值。watcherSetByKey 提供了 add()和 delete()两个方法，分别用来添加和删除

指定的 watcher 实例，其实现比较简单，这里不再赘述，感兴趣的读者可以查看源码进行学习。

- ranges(adt.IntervalTree 类型): 该字段记录进行范围监听的 watcher 实例，IntervalTree（线段树）是二叉树的一种变形，线段树将一个区间划分成一些单元区间，每一个区间对应线段树的一个叶节点。假设在线段树中一个非叶子节点[a，b]，那么它的左儿子节点表示的区间范围为是[a，(a+b)/2]，右儿子节点表示的区间范围为[(a+b)/2+1，b]。线段树的结构大致如图 6-11 所示，其所有叶子节点连接起来就表示整个线段的长度。

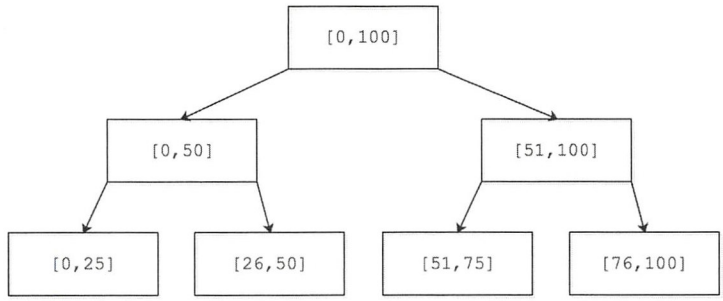


图 6-11

这里不再展开介绍 IntervalTree 的具体实现，需要了解其实现的读者可以参考源码进行学习。

➢ Watch方法

下面介绍 watchableStore 结构体中提供的核心功能。首先是添加 watcher 实例的功能，该功能是在其 watch()方法中实现的，具体实现如下：

```
func (s *watchableStore) watch(key, end []byte, startRev int64, id WatchID,
      ch chan<- WatchResponse, fcs ...FilterFunc) (*watcher, cancelFunc) {
  wa := &watcher{ // 创建 watcher 实例
    key:    key,
    end:    end,
    minRev: startRev,
    id:     id,
    ch:     ch,
    fcs:    fcs,
  }
  // 加 mu 锁，因为要读取 currentRev 字段，所以还需要加 revMu 锁(略)
  // 比较 startRev 与 store.currentRev，决定待添加的 watcher 实例是否已同步完成
  synced := startRev > s.store.currentRev || startRev == 0
  if synced {
```

```
        wa.minRev = s.store.currentRev + 1 // 设置待添加 watcher 的 minRev 字段值
        if startRev > wa.minRev {
            wa.minRev = startRev
        }
    }
    // 如果待添加 watcher 已同步完成，则将其添加到 synced watcherGroup 中，否则添加
    // 到 unsynced watcherGroup 中
    if synced {
        s.synced.add(wa)
    } else {
        s.unsynced.add(wa)
    }
    return wa, func(){s.cancelWatcher(wa)} // 返回该 watcher 实例，以及取消该 watcher 实例
                                           // 的回调函数
}
```

在向（un）synced watcherGroup 中添加 watcher 实例时，都是通过调用 watcherGroup.add() 方法实现的，其具体实现如下：

```
func (wg *watcherGroup) add(wa *watcher) {
    wg.watchers.add(wa) // 首先将 watcher 实例添加到 watchers 字段中保存
    if wa.end == nil { // watcher.end 为空表示只监听单个 Key，则将其添加到 keyWatchers 中保存
        wg.keyWatchers.add(wa)
        return
    }
    // 如果待添加的是范围 watcher，则执行下面的添加逻辑
    // 根据待添加 watcher 的 key 和 end 字段创建 IntervalTree 中的一个节点
    ivl := adt.NewStringAffineInterval(string(wa.key), string(wa.end))
    if iv := wg.ranges.Find(ivl); iv != nil {
        // 如果在 IntervalTree 中查找到了对应节点，则将该 watcher 添加到对应的 watcherSet 中
        iv.Val.(watcherSet).add(wa)
        return
    }
    // 当前 IntervalTree 中没有对应节点，则创建对应的 watcherSet，并添加到 IntervalTree 中
    ws := make(watcherSet)
    ws.add(wa)
    wg.ranges.Insert(ivl, ws)
}
```

➢ cancelWatcher方法

watchableStore.watch() 方法返回的取消 watcher 实例的回调函数是通过调用 watchableStore.cancelWatcher()方法实现的，下面简单看一下该方法的具体实现：

```
func (s *watchableStore) cancelWatcher(wa *watcher) {
  for {
    // 加锁(略)
    if s.unsynced.delete(wa) { // 尝试从 unsynced watcherGroup 中删除该 watcher 实例
      slowWatcherGauge.Dec()
      break
    } else if s.synced.delete(wa) {// 尝试从 synced watcherGroup 中删除该 watcher 实例
      break
    } else if wa.compacted { // 如果待删除的 watcher 实例已经因为压缩操作而删除，则直接返回
      break
    }

    // 如果在(un)synced watcherGroup 中都没有，该 watcher 可能已经被触发了，
    // 则在 victims 中查找
    var victimBatch watcherBatch
    for _, wb := range s.victims {
      if wb[wa] != nil { // 查找待删除 watcher 对应的 watcherBatch 实例
        victimBatch = wb
        break
      }
    }
    if victimBatch != nil {
      slowWatcherGauge.Dec()
      // 删除 victims 字段中对应的 watcher 实例
      delete(victimBatch, wa)
      break
    }
    // 如果未找到，可能是待删除的 watcher 刚刚从 synced watcherGroup 中删除且未添加到 victims
    // 中，所以稍后进行重试
    time.Sleep(time.Millisecond)
  }
}
```

➢ syncWatchersLoop方法

通过前面对 newWatchableStore()函数的介绍我们知道，在初始化 watchableStore 实例时会启动两个后台 goroutine，其中一个 goroutine 执行 watchableStore.syncWatchersLoop()方法，该 goroutine 的主要作用是每隔 100ms 对 unsynced watcherGroup 进行一次批量的同步，其具体实现如下：

```
func (s *watchableStore) syncWatchersLoop() {
  defer s.wg.Done() // 该后台 goroutine 结束时调用，wg 字段的作用在前面介绍过了，这里不再赘述
  for {
    lastUnsyncedWatchers := s.unsynced.size() // 获取当前的 unsynced watcherGroup 的大小
    unsyncedWatchers := 0
    if lastUnsyncedWatchers > 0 {
      // 存在需要进行同步的 watcher 实例，则调用 syncWatchers()方法对 unsynced
      // watcherGroup 中的 watcher 进行批量同步
      unsyncedWatchers = s.syncWatchers()
    }
    waitDuration := 100 * time.Millisecond
    // 调整该后台 goroutine 执行的时间间隔(略)
    select { // 等待 waitDuration 时长之后，再开始下次同步
    case <-time.After(waitDuration):
    case <-s.stopc:
      return
    }
  }
}
```

从上面对 syncWatchersLoop ()方法的分析可以看出，syncWatchers()方法是批量同步 unsynced watcherGroup 的核心，该方法的大致步骤如下：

（1）从 unsynced watcherGroup 中选择一批 watcher 实例，作为此次需要进行同步的 watcher 实例。

（2）从该批 watcher 实例中查找最小的 minRev 字段值。

（3）在 BoltDB 中进行范围查询，查询 minRev～currentRev（当前 revision 值）的所有键值对。

（4）过滤掉该批 watcher 实例中因 minRev～currentRev 之间发生压缩操作而被清除的 watcher 实例。

（5）遍历步骤 3 中查询到的键值对，将其中的更新操作转换为 Event 事件，然后封装成 WatchResponse，并写入对应的 watcher.ch 通道中。如果 watcher.ch 通道被填充满，则将 Event 事件记录到 watchableStore.victims 中，并由另一个后台 goroutine 处理该字段中积累的 Event 事件。

（6）将已经完成同步的 watcher 实例记录到 synced watcherGroup 中，同时将其中 unsynced watcherGroup 中删除。

了解了 watchableStore.syncWatchers()方法的大致步骤之后，再来分析其实现就会相对简单点，如下所示。

```
func (s *watchableStore) syncWatchers() int {
  // 如果 unsynced watcherGroup 为空，则直接返回(略)
  curRev := s.store.currentRev
  compactionRev := s.store.compactMainRev
  // 从 unsynced watcherGroup 中查找一批此次需要同步的 watcher 实例，
  // 并将这些 watcher 实例封装成 watcherGroup 实例返回，返回的 minRev 是这批待同步的
  // watcher 实例中 minRev 字段最小值
  // 后面详细介绍 watcherGroup.choose()方法的具体实现
  wg, minRev := s.unsynced.choose(maxWatchersPerSync, curRev, compactionRev)
  // 将 minRev 和 currentRev 写入 minBytes 和 maxBytes 中
  minBytes, maxBytes := newRevBytes(), newRevBytes()
  revToBytes(revision{main: minRev}, minBytes)
  revToBytes(revision{main: curRev + 1}, maxBytes)

  tx := s.store.b.ReadTx() // 获取只读事务
  tx.Lock()
  // 调用 UnsafeRange()方法，对 key Bucket 进行范围查找
  revs, vs := tx.UnsafeRange(keyBucketName, minBytes, maxBytes, 0)
  // 将查询到的键值对转换成对应的 Event 事件，后面详细介绍 kvsToEvents()函数的实现
  evs := kvsToEvents(wg, revs, vs)
  tx.Unlock()

  var victims watcherBatch
  wb := newWatcherBatch(wg, evs) // 将上述 watcher 集合及 Event 事件封装成 watcherBatch
  for w := range wg.watchers {
    w.minRev = curRev + 1 // 更新 watcher.minRev 字段，这样就不会被同一个修改操作触发两次
    eb, ok := wb[w]
    if !ok {
```

```
            // 该 watcher 实例所监听的键值对没有更新操作（在 minRev～currentRev 之间），
            // 所以没有被触发，故同步完毕，这里会将其转移到 unsynced watcherGroup 中
            s.synced.add(w)
            s.unsynced.delete(w)
            continue
        }
        if eb.moreRev != 0 {
            // 在 minRev～currentRev 之间触发当前 watcher 的更新操作过多，无法全部放到一个
            // eventBatch 中，这里只将该 watcher.minRev 设置成 moreRev，则下次处理 unsynced
            // watcherGroup 时，会将其后的更新操作查询出来继续处理
            w.minRev = eb.moreRev
        }

        // 将前面创建的 Event 事件集合封装成 WatchResponse 实例，然后写入 watcher.ch 通道中
        if w.send(WatchResponse{WatchID: w.id, Events: eb.evs, Revision: curRev}) {
            pendingEventsGauge.Add(float64(len(eb.evs)))
        } else { // 如果 watcher.ch 通道阻塞，则将该 watcher.victim 字段设置为 true
            if victims == nil {
                victims = make(watcherBatch) // 初始化 victims 变量
            }
            w.victim = true
        }

        if w.victim { // 如果 watcher.ch 通道阻塞，则将触发它的 Event 事件记录到 victims 中
            victims[w] = eb
        } else {
            if eb.moreRev != 0 { // 如果后续还有其他未处理的 Event 事件，则当前 watcher 依然未同
                                 // 步完成
                continue
            }
            s.synced.add(w) // 当前 watcher 已经同步完成，加入 synced watcherGroup 中
        }
        s.unsynced.delete(w) // 将当前 watcher 从 unsynced watcherGroup 中删除
    }
    // 将 victims 变量中记录 watcherBatch 添加到 watchableStore.victims 中，等待处理
    s.addVictim(victims)
    return s.unsynced.size() // 返回此次处理之后 unsynced watcherGroup 的长度
}
```

在 watchableStore.syncWatchers()方法中，有几个方法我们需要详细介绍一下，首先是watcherGroup.choose()方法。该方法会根据 unsynced watcherGroup 中记录的 watcher 个数对其进行分批返回。另外，它还会获取该批 watcher 实例中查找最小的 minRev 字段。watcherGroup.choose()方法的具体实现如下所示。

```
func (wg *watcherGroup) choose(maxWatchers int, curRev, compactRev int64)
        (*watcherGroup, int64) {
    // 当前 unsynced watcherGroup 中记录的 watcher 个数未达到指定上限(默认值为 512)，则直接调
    // 用 unsynced watcherGroup 的 chooseAll()方法获取所有未完成同步的 watcher 中最小的 minRev
    // 字段值
    if len(wg.watchers) < maxWatchers {
        return wg, wg.chooseAll(curRev, compactRev)
    }
    // 如果当前 unsynced watcherGroup 中的 watcher 个数超过指定上限，则需要分批处理
    ret := newWatcherGroup() // 创建新的 watcherGroup 实例
    // 从 unsynced watcherGroup 中遍历得到指定量的 watcher，并添加到新建的 watcherGroup 实例中
    for w := range wg.watchers {
        if maxWatchers <= 0 {
            break
        }
        maxWatchers--
        ret.add(w)
    }
    // 依然通过 chooseAll()方法从该批 watcher 实例中查找最小的 minRev 字段
    return &ret, ret.chooseAll(curRev, compactRev)
}
```

紧接着进入 watcherGroup.chooseAll()方法，它会遍历 watcherGroup 中记录的全部 watcher 实例，记录其中最小的 minRev 字段值并返回，其具体实现如下：

```
func (wg *watcherGroup) chooseAll(curRev, compactRev int64) int64 {
    minRev := int64(math.MaxInt64) // 用于记录该 watcherGroup 中最小的 minRev 字段
    for w := range wg.watchers { // 遍历 watcherGroup 中全部 watcher 实例
        if w.minRev < compactRev { // 该 watcher 因为压缩操作而被删除
            select {
            // 创建 watchResponse(注意，其 CompactRevision 字段是有值的)，并写入 watcher.ch 通道
            case w.ch <- WatchResponse{WatchID: w.id, CompactRevision: compactRev}:
```

```
            w.compacted = true // 将 watcher.compacted 设置为 false，表示因压缩操作而被删除
            wg.delete(w) // 将该 watcher 实例从 watcherGroup 中删除
          default:
            // 如果 watcher.ch 通道阻塞，则下次调用 chooseAll() 方法时再尝试重新删除该 watcher 实例
          }
          continue
        }
        if minRev > w.minRev { // 更新 minRev 遍历，记录最小的 watcher.minRev 字段
          minRev = w.minRev
        }
      }
      return minRev
    }
```

在 watchableStore.syncWatchers()方法中另一个需要详细介绍的是 kvsToEvents()函数，它负责将前面从 BoltDB 中查询到的键值对信息转换成相应的 Event 实例，具体实现如下：

```
func kvsToEvents(wg *watcherGroup, revs, vals [][]byte) (evs []mvccpb.Event) {
  for i, v := range vals { // 键值对数据
    var kv mvccpb.KeyValue
    if err := kv.Unmarshal(v); ... // 反序列化得到 KeyValue 实例
    // 在此批处理的所有 watcher 实例中，是否有监听了该 Key 的实例，
    // 需要注意的是，unsynced watcherGroup 中可能同时记录了监听单个 Key 的 watcher 和
    // 监听范围的 watcher，所以在 watcherGroup.contain() 方法中会同时查找这两种
    if !wg.contains(string(kv.Key)) {
      continue
    }
    ty := mvccpb.PUT // 默认是更新操作
    if isTombstone(revs[i]) {
      ty = mvccpb.DELETE // 如果是 tombstone，则表示删除操作
      kv.ModRevision = bytesToRev(revs[i]).main
    }
    // 将该键值对转换成对应的 Event 实例，并记录下来
    evs = append(evs, mvccpb.Event{Kv: &kv, Type: ty})
  }
  return evs
}
```

syncWatchersLoop()方法在完成 Event 实例的转换之后，会将 Event 实例封装成 watchResponse 实例，并调用 watcher.send()方法将其写入 watcher.ch 通道中。在 watcher.send()方法中会使用 watcher.fcs 中记录的过滤器对 Event 进行过滤，只有通过过滤的 Event 实例才能被放出去，具体实现如下所示。

```
func (w *watcher) send(wr WatchResponse) bool {
  progressEvent := len(wr.Events) == 0 // 发送 watchResponse 中封装的 Event 事件
  if len(w.fcs) != 0 { // 前面介绍过，watcher.fcs 字段中记录了过滤 Event 实例的过滤器
    ne := make([]mvccpb.Event, 0, len(wr.Events))
    for i := range wr.Events { // 遍历 watchResponse 中封装的 Event 实例，并逐个进行过滤
      filtered := false
      for _, filter := range w.fcs {
        if filter(wr.Events[i]) {
          filtered = true
          break
        }
      }
      if !filtered { // 通过所有过滤器检测的 Event 实例才能被记录到 watchResponse 中
        ne = append(ne, wr.Events[i])
      }
    }
    wr.Events = ne
  }
  // 如果没有通过过滤的 Event 实例，则直接返回 true，此次发送结束(略)
  select {
  // 将 watchResponse 实例写入 watcher.ch 通道中，如果 watcher.ch 通道阻塞，
  // 则通过返回 false 进行表示
  case w.ch <- wr:
    return true
  default:
    return false
  }
}
```

如果上述 watcher.ch 通道阻塞了，则会将 watchResponse 中记录的 Event 实例组织成 watcherBatch，并添加到 watchableStore.victims 字段中等待处理。另外会向 watchableStore.victimc 通道中发送一个信号，通知其他后台 goroutine 来处理 victims 字段中积累的 watcherBatch 实例。

这里简单看一下 watchableStore.addVictim()方法的实现：

```
func (s *watchableStore) addVictim(victim watcherBatch) {
  // 将 watcherBatch 实例追加到 watchableStore.victims 字段中
  s.victims = append(s.victims, victim)
  select {
  case s.victimc <- struct{}{}: // 向 victimc 通道中发送一个信号
  default:
  }
}
```

➢ syncVictimsLoop方法

在初始化 watchableStore 实例时启动的另一个后台 goroutine 会执行 syncVictimsLoop()方法，在该方法中会定期处理 watchableStore.victims 中缓存的 watcherBatch 实例。syncVictimsLoop()方法的具体实现如下：

```
func (s *watchableStore) syncVictimsLoop() {
  defer s.wg.Done() // 该后台 goroutine 结束时调用，wg 字段的作用在前面介绍过了，这里不再赘述
  for {
    for s.moveVictims() != 0 { // 循环处理 victims 中缓存的 watcherBatch 实例
    }
    // 根据 victims 的长度决定下次处理 victims 的时间间隔
    select {
    case <-tickc:
    case <-s.victimc:
    case <-s.stopc:
      return
    }
  }
}
```

watchableStore.moveVictims()方法是处理 victims 的核心，该方法会遍历 victims 字段中记录的 watchBatch 实例，并尝试将其中的 Event 实例封装成 watchResponse 重新发送。如果发送依然失败，则将其放回 victims 字段中保存，等待下一次重试；如果发送成功，则根据相应的 watcher 的同步情况，将 watcher 实例迁移到（un）synced watcherGroup 中。watchableStore.moveVictims()方法的具体实现如下：

```
func (s *watchableStore) moveVictims() (moved int) {
```

```
victims := s.victims
s.victims = nil // 清空 victims 字段

var newVictim watcherBatch
for _, wb := range victims { // 遍历 victims 中记录的 watcherBatch
  // try to send responses again
  for w, eb := range wb { // 从 watcherBatch 中获取 eventBatch 实例
    // watcher has observed the store up to, but not including, w.minRev
    rev := w.minRev - 1
    // 将 eventBatch 封装成 watchResponse，并调用 watcher.send()方法尝试将其发送出去
    if w.send(WatchResponse{WatchID: w.id, Events: eb.evs, Revision: rev}) {
    } else { // 如果 watcher.ch 通道依然阻塞，则将对应的 Event 实例重新放回 newVictim 变
             // 量中保存
      if newVictim == nil {
        newVictim = make(watcherBatch)
      }
      newVictim[w] = eb
      continue
    }
    moved++ // 记录重试成功的 eventBatch 个数
  }

  curRev := s.store.currentRev
  for w, eb := range wb { // 遍历 watcherBatch
    // 如果 eventBatch 实例被记录到 newVictim 中，则表示 watcher.ch 通道依然阻塞，
    // watchResponse 发送失败
    if newVictim != nil && newVictim[w] != nil {
      continue
    }
    // 如果 eventBatch 实例未被记录到 newVictim 中，则表示 watchResponse 发送成功
    w.victim = false
    if eb.moreRev != 0 {// 检测当前 watcher 后续是否还有未同步的 Event 事件
      w.minRev = eb.moreRev
    }
    if w.minRev <= curRev { // 当前 watcher 未完成同步，移动到 unsynced watcherGroup 中
      s.unsynced.add(w)
    } else { // 当前 watcher 已经完成同步，移动到 synced watcherGroup 中
      s.synced.add(w)
```

```
            }
        }
    }
    if len(newVictim) > 0 { // 使用 newVictim 更新 watchableStore.victims 字段
        s.victims = append(s.victims, newVictim)
    }
    return moved // 返回重试成功的 eventBatch 个数
}
```

➢ Write与notify

前面介绍 watchableStore 时提到其内嵌了*store，并且在 watchableStore 中重写了 Write()方法。在 watchableStore.Write()方法中返回的是 watchableStoreTxnWrite 实例，watchableStoreTxnWrite 中封装了前面介绍的 storeTxnWrite 和关联的 watchableStore 实例。watchableStoreTxnWrite 重写了 End()方法，如果当前读写事务中存在更新操作，则会在 End()方法中触发相应的 watcher 实例，具体实现如下：

```
func (tw *watchableStoreTxnWrite) End() {
    changes := tw.Changes() // 当前读写事务中执行的更新操作
    if len(changes) == 0 { // 如果当前事务中没有任何更新操作，则直接结束当前事务即可
        tw.TxnWrite.End()
        return
    }

    rev := tw.Rev() + 1
    evs := make([]mvccpb.Event, len(changes))
    for i, change := range changes { // 将当前事务中的更新操作转换成对应的 Event 事件
        evs[i].Kv = &changes[i]
        if change.CreateRevision == 0 {
            evs[i].Type = mvccpb.DELETE
            evs[i].Kv.ModRevision = rev
        } else {
            evs[i].Type = mvccpb.PUT
        }
    }
    // 省略加锁、解锁的代码(略)
    tw.s.notify(rev, evs) // 调用 watchableStore.notify()方法，将上述 Event 事件发送出去
    tw.TxnWrite.End() // 结束当前读写事务
}
```

接着来看一下 watchableStore.notify()方法是如何发送上述 Event 事件的，其具体实现如下：

```
func (s *watchableStore) notify(rev int64, evs []mvccpb.Event) {
  var victim watcherBatch
  // 将传入的 Event 实例转换成 watcerBatch(map[*watcher]eventBatch)，之后进行遍历，
  // 逐个 watcher 进行处理
  for w, eb := range newWatcherBatch(&s.synced, evs) {
    // 将当前 watcher 对应的 Event 实例封装成 watchResponse 实例，并尝试写入 watcher.ch 通道中
    if w.send(WatchResponse{WatchID: w.id, Events: eb.evs, Revision: rev}) {
      pendingEventsGauge.Add(float64(len(eb.evs)))
    } else { // 如果 wathcer.ch 通道阻塞，则将这些 Event 实例记录到 watchableStore.victims
             // 字段中
      // move slow watcher to victims
      w.minRev = rev + 1
      if victim == nil {
        victim = make(watcherBatch)
      }
      w.victim = true
      victim[w] = eb
      s.synced.delete(w) // 将该 watcher 从 synced watcherGroup 中删除
    }
  }
  s.addVictim(victim) // addVictim()方法的具体实现在前面介绍过了，这里不再赘述
}
```

3. WatchStream

前面介绍了 watchableStore 提供的主要功能，watchableStore 实现了 Watchable 接口，该接口中定义了 NewWatchStream()方法，该方法会新建并返回 watchStream 实例。结构体 watchStream 实现了 WatchStream 接口，该接口中定义的具体方法在下面介绍 watchStream 的功能时会详细介绍。

下面先看一下结构体 watchStream 中各个字段的含义：

- watchable（watchable 类型）：用来记录关联的 watchableStore 实例。
- ch（chan WatchResponse 类型）：通过该 watchStream 实例创建的 watcher 实例在被触发时，都会将 Event 事件写入该通道中。
- nextID（WatchID 类型）：在当前 watchStream 实例中添加 watcher 实例时，会为其分配唯一标识，该字段就用来生成该唯一标识。

- watchers (map[WatchID]*watcher 类型): 该字段记录唯一标识与对应 watcher 实例之间的映射关系。
- cancels (map[WatchID]cancelFunc 类型): 该字段记录唯一标识与取消对应 watcher 的回调函数之间的映射关系。

watchStream.Watch()方法中会创建 watcher 实例监听指定的 Key（或是范围监听[key,end)），其中 startRev 参数则指定了该 watcher 实例监听的起始 revision，如果 startRev 参数小于等于 0，则表示从当前 revision 开始监听，fcs 参数则是前面介绍的 Event 事件过滤器。watchStream.Watch()方法的具体实现如下：

```
    func (ws *watchStream) Watch(key, end []byte, startRev int64, fcs ...FilterFunc)
WatchID {
        // 检测参数的合法性，检测当前 watchStream 是否已经关闭(略)
        id := ws.nextID // 生成新建 watcher 实例的唯一标识
        ws.nextID++
        // 调用 watchableStore.watcher()方法创建 watcher 实例，注意，新建的 watcher 实例共用了同一
        // 个通道
        // 该方法在前面已经详细介绍过了，这里不再赘述
        w, c := ws.watchable.watch(key, end, startRev, id, ws.ch, fcs...)
        ws.watchers[id] = w // 记录新建的 watcher 实例
        ws.cancels[id] = c // 保存取消 watcher 实例时用到的回调函数
        return id
    }
```

watchStream.Cancel()方法中会查找并调用指定 watcher 实例对应的取消回调函数，具体实现如下：

```
    func (ws *watchStream) Cancel(id WatchID) error {
        cancel, ok := ws.cancels[id] // 根据 watcher 唯一标识，查找对应的取消回调函数
        ok = ok && !ws.closed
        if ok {
            delete(ws.watchers, id) // 从 watchStream.watchers 字段中清理对应的 watcher 实例
            delete(ws.cancels, id) // 从 watchStream.cancels 中清理对应的取消回调函数
        }
        cancel() // 调用回调函数，取消 watcher 实例，该回调函数在前面详细介绍过了，这里不再赘述
        return nil
    }
```

watchStream.RequestProgress()方法用来检测指定 watcher 的处理进度（progress，即当前 watcher 正在处理哪个 revision 中的更新操作）。只有当 watcher 完全同步时，调用该方法才会创建一个空的 watchResponse，并写入 watcher.ch 通道中，其具体实现如下：

```
func (ws *watchStream) RequestProgress(id WatchID) {
    w, ok := ws.watchers[id] // 从 watchStream.watchers 中查询指定的 watcher 实例
    if !ok {
        return
    }
    ws.watchable.progress(w) // 调用 watchableStore.progress()方法
}

func (s *watchableStore) progress(w *watcher) {
    // 只有 watcher 在 synced watcherGroup 中时，才会响应该方法
    if _, ok := s.synced.watchers[w]; ok {
        // 注意这里发送的 watchResponse 实例不包含任何 Event 事件，只包含当前的 revision 值
        // 如果当前 watcher.ch 通道已经阻塞，其中阻塞的 watchResponse 实例中也包含了 revision 值，
        // 也可以说明当前 watcher 的处理进度，则没有必要继续添加空的 watchResponse 实例了
        w.send(WatchResponse{WatchID: w.id, Revision: s.rev()})
    }
}
```

到此为止，etcd v3 版本存储中的 watcher 机制的原理及具体实现就介绍完了。

6.2.3 Lessor

在前面介绍 etcd v2 的存储时我们看到，非永久节点中都记录了节点的存活时间（ExpireTime），并且通过 ttlKeyHeap 将节点按照过期时间进行了排序。在下一章介绍 etcd 服务端时会看到，服务端会定时发送 SYNC 请求，该请求主要用于触发 v2 存储调用前面分析的 DeleteExpiredKeys()方法清理过期的节点。

在 etcd v3 存储中并没有提供类似的过期时间的设置，而是通过 Lease（租约）实现键值对过期的效果。在 Lessor 接口中定义了授权、撤销、修改 Lease 的相关方法，其具体定义如下所示。

```
type Lessor interface {
    // 创建 Lease 实例，该 Lease 会在指定的时间(ttl)之后过期
    Grant(id LeaseID, ttl int64) (*Lease, error)
    // 将指定 Lease 与指定 LeaseItem 绑定，在 LeaseItem 中封装了键值对的 Key 值
```

```
    Attach(id LeaseID, items []LeaseItem) error
    // 取消指定 Lease 与指定 LeaseItem 之间的绑定关系
    Detach(id LeaseID, items []LeaseItem) error
    // 撤销指定的 Lease，该 Lease 实例相关的 LeaseItem 也会被删除
    Revoke(id LeaseID) error
    // 根据 LeaseItem 查询对应 Lease 实例的 id
    GetLease(item LeaseItem) LeaseID
    // 如果当前节点成为 Leader 节点，则其使用的 Lessor 实例将通过该方法晋升，成为主 Lessor
    // 主 Lessor 将控制着整个集群中所有 Lease 实例
    Promote(extend time.Duration)
    // 如果当前节点从 Leader 状态转换为其他状态，则会通过该方法将其使用的 Lessor 实例进行降级
    Demote()
    // 续约指定的 lease 实例
    Renew(id LeaseID) (int64, error)
    // 查找指定 id 对应的 Lease 实例
    Lookup(id LeaseID) *Lease
    // 如果出现 lease 实例过期，则会被写入到 ExpiredLeasesC 通道中
    ExpiredLeasesC() <-chan []*Lease
    // 从底层的 v3 存储中恢复 Lease 实例
    Recover(b backend.Backend, rd RangeDeleter)
}
```

Lessor 接口有两个实现，从名字也可以看出，FakeLessor 实现是一个空实现，如图 6-12 所示。这里重点介绍 lessor 实现，其中各个字段的含义如下所示。

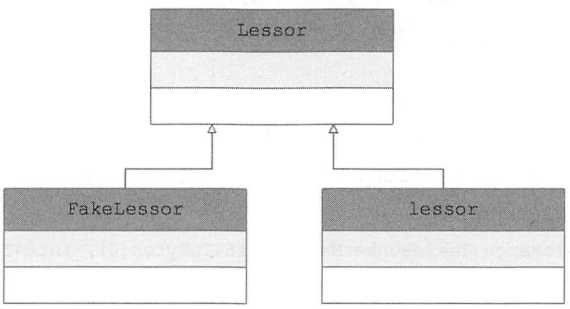


图 6-12

- leaseMap（map[LeaseID]*Lease 类型）：记录 id 到 Lease 实例之间的映射。在 Lease 结构体中记录了如下内容。
  - ID（int64 类型）：该 Lease 实例的唯一标识。

  - ttl（int64 类型）：该 Lease 实例的存活时长。
  - expiry（monotime.Time 类型）：该 Lease 实例过期的时间戳。
  - itemSet（map[LeaseItem]struct{}类型）：该 map 中的 Key 是与当前 Lease 实例绑定的 LeaseItem 实例，Value 始终为空结构体。
  - revokec（chan struct{}类型）：该 Lease 实例被撤销时会关闭该通道，从而实现通知监听该通道的 goroutine 的效果。
- itemMap（map[LeaseItem]LeaseID 类型）：记录 LeaseItem 到 Lease 实例 id 的映射。
- rd（RangeDeleter 类型）：RangeDeleter 接口主要用于从底层的存储中删除过期的 lease 实例。
- b（backend.Backend 类型）：底层持久化 Lease 的存储，具体持久化哪些信息，在后面会详细介绍。
- minLeaseTTL（int64 类型）：Lease 实例过期时间的最小值。
- expiredC（chan []*Lease 类型）：过期的 Lease 实例会被写入该通道中，并等待其他 goroutine 进行处理。
- demotec（chan struct{}类型）：用于判断当前 lessor 实例是否为主 lessor（primary lessor）。如果当前 lessor 是主 Lessor 实例，则会开启该通道，当节点切换成非 Leader 状态时，会关闭该通道。

1. 初始化

在 newLessor()函数中会创建 lessor 实例，并调用 initAndRecover()方法完成初始化操作，同时还会启动一个后台 goroutine 查找当前 lessor 中是否存在过期的 lease 实例。下面我们先来看一下 lessor.initAndRecover()方法的具体实现，如下所示。

```
func (le *lessor) initAndRecover() {
  tx := le.b.BatchTx() // 开启 v3 存储的读写事务，省略后续加锁解锁的相关代码
  tx.UnsafeCreateBucket(leaseBucketName) // 创建 lease Bucket
  // 查找 lease Bucket 中全部信息
  _, vs := tx.UnsafeRange(leaseBucketName, int64ToBytes(0), int64ToBytes(math.MaxInt64),
    0)
  for i := range vs {
    var lpb leasepb.Lease
    // 反序列化得到 leasepb.Lease 实例，其中只记录了对应的 id 和过期时间，错误处理（略）
    err := lpb.Unmarshal(vs[i])
    ID := LeaseID(lpb.ID)
```

```
        if lpb.TTL < le.minLeaseTTL { // 修正过期时间
            lpb.TTL = le.minLeaseTTL
        }
        le.leaseMap[ID] = &Lease{ // 创建 Lease 实例并记录到 leaseMap 中
            ID:  ID,
            ttl: lpb.TTL,
            itemSet: make(map[LeaseItem]struct{}),
            expiry:  forever,
            revokec: make(chan struct{}),
        }
    }
    le.b.ForceCommit() // 提交事务
}
```

在 newLessor()函数中启动的后台 goroutine 执行的是 lessor.runLoop()方法，其中会根据当前 lessor 实例是否为主 lessor，决定是否检测过期 Lease 实例，其具体实现如下所示。

```
func (le *lessor) runLoop() {
    for {
        var ls []*Lease
        if le.isPrimary() { // 检测当前的 lessor 是否为主 Lessor
            ls = le.findExpiredLeases() // 查询
        }

        if len(ls) != 0 {
            if len(ls) > leaseRevokeRate/2 {  // 如果过期的 Lease 实例过多，则进行限流
                ls = ls[:leaseRevokeRate/2]
            }
            select {
            case le.expiredC <- ls: // 将过期的 Lease 实例集合
            default: // 如果当前 expiredC 通道阻塞，则放弃处理检测
            }
        }
        select {
        case <-time.After(500 * time.Millisecond): // 阻塞 500 毫秒，再开始下次检测
        // 省略当前 lessor 实例关闭的处理
        }
    }
}
```

在 lessor.findExpiredLeases()方法中会遍历 leaseMap 中全部的 Lease 实例，并调用 Lease.expired()方法检测其是否过期，具体实现如下所示。

```
func (le *lessor) findExpiredLeases() []*Lease {
    leases := make([]*Lease, 0, 16) // 记录过期的 Lease 实例
    for _, l := range le.leaseMap {
        if l.expired() { // 检测该 Lease 实例是否过期，其中通过当前时间与 Lease.expiry 进行比较
            leases = append(leases, l)
        }
    }
    return leases
}
```

2. Grant 方法

lessor.Grant()方法会根据指定的 id 和过期时长新建 Lease 实例，然后将其保存到 leaseMap 中并进行持久化，其具体实现如下所示。

```
func (le *lessor) Grant(id LeaseID, ttl int64) (*Lease, error) {
    l := &Lease{ // 新建 Lease 实例
        ID:      id,
        ttl:     ttl,
        itemSet: make(map[LeaseItem]struct{}),
        revokec: make(chan struct{}),
    }
    // 检测指定的 id 在当前 leaseMap 中是否存在对应 Lease 实例(略)
    // 修正过期时长(略)
    if le.isPrimary() {
        l.refresh(0) // 更新 Lease 实例的过期时间戳
    } else {
        l.forever() // 如果当前节点不是 Leader，则将 Lease 设置为永不过期
    }
    le.leaseMap[id] = l // 在 leaseMap 中记录该 Lease 实例
    l.persistTo(le.b) // 持久化 Lease 信息
    return l, nil
}
```

3. Revoke 方法

lessor.Revoke()方法负责撤销指定 id 对应的 Lease 实例，其中会关闭 Lease 实例对应的 revokec 通道，并将 Lease 实例从 lessor.leaseMap 和存储中删除。需要注意的是，除了删除 Lease 实例本身，还需要删除与该 Lease 实例关联的键值对。lessor.Revoke()方法的具体实现如下所示。

```
func (le *lessor) Revoke(id LeaseID) error {
    l := le.leaseMap[id] // 从 leaseMap 中查找指定的 Lease 实例
    defer close(l.revokec) // 在方法结束时，关闭该 Lease 实例中的 revokec 通道
    txn := le.rd()
    keys := l.Keys() // 获取与该 Lease 实例绑定的 Key 值
    sort.StringSlice(keys).Sort()
    for _, key := range keys {
        txn.DeleteRange([]byte(key), nil) // 删除与该 Lease 实例绑定的全部键值对
    }
    delete(le.leaseMap, l.ID) // 将该 Lease 实例从 leaseMap 中删除
    // 将该 Lease 实例的信息从底层的 v3 存储中删除
    le.b.BatchTx().UnsafeDelete(leaseBucketName, int64ToBytes(int64(l.ID)))
    txn.End() // 提交事务
    return nil
}
```

4. Renew 方法

lessor.Renew()方法负责续租一个已经存在的 Lease 实例，当然，只有当前节点是 Leader 时才能完成该续租操作。Renew()方法的具体实现如下所示。

```
func (le *lessor) Renew(id LeaseID) (int64, error) {
    // 检测当前 lessor 是否为主 lessor 实例(略)
    l := le.leaseMap[id] // 在 leaseMap 中查找指定的 Lease 实例
    // 如果指定 Lease 不存在，则返回错误信息(略)
    if l.expired() { // 检测 Lease 实例是否过期
        // 如果该 Lease 已经过期，则返回错误信息(略)
    }
    l.refresh(0) // 更新过期时间
    return l.ttl, nil
}
```

5. Attach&Detach 方法

lessor.Attach()方法负责将指定的 Lease 实例和键值对绑定，Detach()方法负责解绑指定的 Lease 实例和键值对解绑。这里简单介绍 Attach()方法，Detach()方法留给读者自己分析。lessor.Attach()方法的具体实现如下：

```
func (le *lessor) Attach(id LeaseID, items []LeaseItem) error {
  l := le.leaseMap[id] // 从 leaseMap 中查找指定的 Lease 实例，错误处理(略)
  for _, it := range items {
    l.itemSet[it] = struct{}{} // 在 itemSet 这个 map 中记录该 LeaseItem 实例
    le.itemMap[it] = id // 在 Lease 实例的 itemMap 字段中记录该 LeaseItem 实例，从而实现绑定
  }
  return nil
}
```

6. v3 存储中的应用

在前面介绍 etcd v3 存储时省略了 Lease 相关的代码，这里回过头来简单分析一下。先来看一下 storeTxnWrite.Put()方法，它在执行更新操作之前会先将 Key 值封装成 LeaseItem 实例，之后通过 lessor.GetLease()方法查找对应的 Lease 实例，然后完成对键值对的更新操作，最后将更新的键值对与原来的 Lease 实例解绑，并与新指定的 Lease 实例绑定。相关的代码片段如下所示。

```
func (tw *storeTxnWrite) put(key, value []byte, leaseID lease.LeaseID) {
  oldLease := lease.NoLease
  // 在内存索引中查找对应的键值对信息(略)
  if err == nil { // 记录键值对绑定的 Lease 实例
    oldLease = tw.s.le.GetLease(lease.LeaseItem{Key: string(key)})
  }
  // 在前面已经分析过具体的更新逻辑，这里不再展开(略)
  if oldLease != lease.NoLease { // 将键值对与之前的 Lease 实例解绑
    err = tw.s.le.Detach(oldLease, []lease.LeaseItem{{Key: string(key)}})
  }
  if leaseID != lease.NoLease { // 将键值对与新指定的 Lease 实例绑定
    err = tw.s.le.Attach(leaseID, []lease.LeaseItem{{Key: string(key)}})
  }
}
```

在 storeTxnWrite.Delete()方法中除了添加 Tombstone，还会解绑键值对与 Lease 实例，具体

实现如下所示。

```
func (tw *storeTxnWrite) delete(key []byte, rev revision) {
  // 添加 Tombstone 的具体逻辑在前面已经详细分析过了，这里不再展开介绍
  item := lease.LeaseItem{Key: string(key)} // 创建 LeaseItem 实例
  leaseID := tw.s.le.GetLease(item) // 根据 LeaseItem 查找对应的 Lease 实例
  if leaseID != lease.NoLease {
    err = tw.s.le.Detach(leaseID, []lease.LeaseItem{item}) // 调用 Detach()方法解绑
  }
}
```

最后看一下 store.restore()方法，其中除了恢复内存索引，还会重新为键值对绑定相关的 Lease 实例。这里读者可以先回顾一下 restoreChunk ()方法，其中会将键值对和 Lease 实例的绑定关系记录到 keyToLease 这个 map 中，其大致代码片段如下：

```
func restoreChunk(kvc chan<- revKeyValue, keys, vals [][]byte,
      keyToLease map[string]lease.LeaseID) {
  for i, key := range keys {
    rkv := revKeyValue{key: key}
    if err := rkv.kv.Unmarshal(vals[i]); err != nil...
    rkv.kstr = string(rkv.kv.Key)
    if isTombstone(key) { // 如果是 Tombstone，则从 keyToLease 中删除
      delete(keyToLease, rkv.kstr)
    } else if lid := lease.LeaseID(rkv.kv.Lease); lid != lease.NoLease {
      keyToLease[rkv.kstr] = lid  // 如果键值对有绑定的 Lease 实例，则记录到 keyToLease 中
    } else {
      delete(keyToLease, rkv.kstr) // 如果该键值对没有绑定 Lease，则从 keyToLease 中删除
    }
    kvc <- rkv
  }
}
```

在 restore()方法中就会应用 keyToLease 这个 map 中记录的对应关系，完成相应的恢复操作，相关的代码实现如下：

```
func (s *store) restore() error {
  keyToLease := make(map[string]lease.LeaseID)
  // 省略其他与 Lease 不相关的代码
```

```
    for {
        // 查询 restoreChunkKeys 数量的键值对
        keys, vals := tx.UnsafeRange(keyBucketName, min, max, int64(restoreChunkKeys))
        restoreChunk(rkvc, keys, vals, keyToLease) // 调用 restoreChunk()方法
    }
    // 省略其他与 Lease 不相关的代码
    for key, lid := range keyToLease {
        err := s.le.Attach(lid, []lease.LeaseItem{{Key: key}}) // 绑定键值对与指定的 Lease 实例
    }
    return nil
}
```

本章小结

本章主要介绍了 etcd v2 版本存储和 etcd v3 版本存储的原理及具体实现。etcd v2 版本存储是完全的内存实现，其数据以树形结构的方式维护在内存中。其持久化方式是将整个存储的数据序列化成 JSON 格式的数据，并写入磁盘文件中。

v2 版本存储中两个比较明显的弊端：一是 EventHistroy 表的长度限制，这会导致更新事件的丢失；二是如果 watcher 被触发后发送事件时出现了阻塞，则 etcd 会直接把 watcher 删除，并需要客户端重新连接。尽管如此，v2 版本存储并没有被 v3 版本存储完全替换，etcd 服务端在某些场景下还是会使用 v2 版本存储，在后面分析服务端实现时还会见到其身影。

在 v3 版本的存储中，键值对数据是持久化到 BoltDB 中，在 BoltDB 中记录的 Key 是 revision 信息（由 main revision 和 sub revision 两部分构成），Value 是 KeyValue 实例序列化之后的内容，在 KeyValue 中包含了客户端发送的原始 Key 和 Value 值。为了加快查询，v3 版本存储使用 Google 开源的 BTree 实现建立了内存索引，其中维护了原始 Key 值与 revision 信息之间的映射关系。在客户端查询键值对时，就可以先查询内存中的 BTree 索引，将原始的 Key 值转换成对应的 revision 信息，然后通过 revision 查询 BoltDB 中保存的真正的键值对数据。另外，v3 存储也支持前缀方式的查询和范围查询。

etcd v3 存储的 watcher 机制是基于上述存储方式的，在 watchableStore 中维护了 synced watcherGroup 和 unsynced watcherGroup，这两个集合分别用于记录已经完全同步的 watcher 实例和未完全同步的 watcher 实例。当 watcher 实例被触发时，会发送相应的 Event 事件，如果发送出现阻塞（例如，网络阻塞或是客户端读取慢等情况），则会将其记录到 watchableStore.victims 中。而 watchableStore 会启动两个后台 goroutine，一个负责定时同步 unsynced watcherGroup 中的 watcher 实例，另一个负责重新发送 watchableStore.victims 中积累的 Event 事件。

仅供非商业用途或交流学习使用

v3 存储通过前面介绍的 synced watcherGroup、synced watcherGroup 和 backend 的持久化存储，可以支持从任意版本开始监听数据，从而解决了 v2 存储中 EventHistroy 表的长度限制所带来的问题。当然，这是在没有压缩操作的前提下，如果出现压缩操作，则会清理掉指定 revision 之前的版本，也就无法对这些版本的修改进行监听了。另外，v3 存储通过 watchableStore.victims 缓存暂时发送失败的 Event 事件，并在稍后通过后台 goroutine 重试，也在一定程度上解决了 v2 版本中第二个弊端。虽然在 etcd v3 存储中废弃了树形的存储结构，但是我们依然可以通过前缀的方式，模拟出与 v2 版本存储类似的树形结构。

最后，希望通过本章的描述，读者能够了解 etcd v2 和 v3 版本存储的原理和大致实现，能够了解 v2 版本存储有哪些问题，以及 v3 版本存储是如何解决的，也为后面分析 etcd 服务端打下基础。

仅供非商业用途或交流学习使用



第 7 章 etcd-server 详解

7.1 raftNode结构体

通过前面对 raftexample 示例的介绍我们知道，在该示例中有一个名为 raftNode 的结构体。在 etcd-server 模块中，有一个与之同名的结构体，两者的主要作用也非常相似，都是充当 etcd-raft 模块与上层模块之间交互的桥梁。如果没有特殊说明，则本章中所说的 raftNode 就是指 etcd-server 模块中定义的 raftNode 结构体。

下面先来看一下 raftNode 中各个字段的含义。

- term（uint64 类型）：当前节点已应用 Entry 记录的最大任期号，在后面分析的过程中我们会看到，对 term、index 和 lead 三个字段的读写都是原子操作。
- index（uint64 类型）：当前节点中已应用 Entry 记录的最大索引值。
- lead（uint64 类型）：记录当前集群中 Leader 节点的 ID 值。
- msgSnapC（chan raftpb.Message 类型）：在前面分析中提到，etcd-raft 模块通过返回 Ready 实例与上层模块进行交互，其中 Ready.Message 字段记录了待发送的消息，其中可能会包含 MsgSnap 类型的消息，该类型消息中封装了需要发送到其他节点的快照数据。当 raftNode 收到 MsgSnap 消息之后，会将其写入 msgSnapC 通道中，并等待上层模块进行发送。
- applyc（chan apply 类型）：正如前文所述，在 etcd-raft 模块返回的 Ready 实例中，除了封装了待持久化的 Entry 记录和待持久化的快照数据，还封装了待应用的 Entry 记录。raftNode 会将待应用的记录和快照数据封装成 apply 实例（apply 的相关内容后面的章

仅供非商业用途或交流学习使用

节中会详细分析），之后写入 applyc 通道等待上层模块处理。

- readStateC (chan raft.ReadState 类型)：Readyc.ReadStates 中封装了只读请求相关的 ReadState 实例，其中的最后一项将会被写入 readStateC 通道中等待上层模块处理。
- ticker (*time.Ticker 类型)：该定时器就是逻辑时钟，每触发一次就会推进一次底层的选举计时器和心跳计时器。
- td(*contention.TimeoutDetector 类型):检测发往同一节点的两次心跳消息是否超时，如果超时，则会打印相关警告信息。

另外需要读者注意的是，结构体 raftNode 中内嵌了 raftNodeConfig，而在结构体 raftNodeConfig 中又内嵌了前面介绍的 etcd-raft 模块中的 Node。这样 raftNode 实例就可以与 etcd-raft 模块完成交互。结构体 raftNodeConfig 中各字段的含义如下。

- isIDRemoved (func(id uint64) bool 类型)：该函数用来检测指定节点是否已经被移出当前集群。
- raftStorage (*raft.MemoryStorage 类型)：与前面介绍的 raftLog.storage 字段指向的 MemoryStorage 为同一实例，主要用来保存持久化的 Entry 记录和快照数据。
- storage (etcdserver.Storage 类型)：注意该字段的类型，在 etcd-raft 模块中有一个与之同名的接口（raft.Storage 接口），MemoryStorage 就是 raft.Storage 接口的实现之一。如果没有特殊说明，在本章中提到的 Storage 接口都是指 etcdserver.Storage。etcdserver.Storage 接口的定义如下：

```
type Storage interface {
  // Save()方法负责将 Entry 记录和 HardState 状态信息保存到底层的持久化存储上，该方法可能
  // 会阻塞，在后面的介绍中会看到，Storage 接口的实现是通过 WAL 模块将上述数据持久化到 WAL
  // 日志文件中的
  Save(st raftpb.HardState, ents []raftpb.Entry) error

  // SaveSnap()方法负责将快照数据持久到底层的持久化存储上，该方法也可能会阻塞，在后面分析中
  // 会看到，Storage 接口的实现是使用前文介绍的 Snapshotter 将快照数据保存到快照文件中的
  SaveSnap(snap raftpb.Snapshot) error
}
```

- transport (rafthttp.Transporter 类型)：通过网络将消息发送到集群中其他节点，Transporter 的工作原理及其相关实现在前面已经详细分析过了，这里不再重复介绍，读者可以回顾前面的章节。

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

- heartbeat（time.Duration 类型）：逻辑时钟的刻度，在后面介绍 raftNode 实例的初始化过程时，还会提到该字段的作用。

介绍完 raftNode 结构体中各个字段的含义及其相关组件的结构之后，我们接下来分析 newRaftNode()函数，其参数是前面介绍的 raftNodeConfig 实例，该函数的主要功能就是创建上面介绍的各种通道，以及创建 raftNode 实例，其具体实现如下：

```
func newRaftNode(cfg raftNodeConfig) *raftNode {
  r := &raftNode{
    raftNodeConfig: cfg,
    td:        contention.NewTimeoutDetector(2 * cfg.heartbeat),
    readStateC: make(chan raft.ReadState, 1),
    msgSnapC:   make(chan raftpb.Message, maxInFlightMsgSnap), // 注意该通道的缓冲区大小
    applyc:     make(chan apply),
    ... ...
  }
  if r.heartbeat == 0 { // 根据 raftNodeConfig.hearbeat 字段创建逻辑时钟，其时间刻度是
                        // heartbeat
    r.ticker = &time.Ticker{}
  } else {
    r.ticker = time.NewTicker(r.heartbeat)  // 每隔 heartbeat 触发一次
  }
  return r
}
```

start 方法详解

完成 raftNode 实例的创建之后，就可以通过 raftNode.start()方法启动相关服务，在该方法中会启动一个独立的后台 goroutine，在该后台 goroutine 中完成了绝大部分与底层 etcd-raft 模块交互的功能，其大致实现如下：

```
func (r *raftNode) start(rh *raftReadyHandler) {
  go func() { // 启动后台的 goroutine 提供服务
    defer r.onStop()
    islead := false  // 刚启动时会将当前节点标识为 follower
    for {
        select {
        case <-r.ticker.C: // 计时器到期被触发，调用 Tick()方法推进选举计时器和心跳计时器
          r.Tick()
```

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用


```
            case rd := <-r.Ready():
                ... ... // 整个 Ready 实例的处理过程比较长，其具体实现下面会详细分析
                // 最后调用 raft.node.Advance()方法，通知 etcd-raft 模块此次 Ready 实例已经处理
                // 完成，etcd-raft 模块更新相应信息(例如，已应用 Entry 的最大索引值)之后，可以继
                // 续返回 Ready 实例
                r.Advance()
            case <-r.stopped:
                return // 当前节点已停止

        }
    }()
}
```

图 7-1 展示了 raftNode 对 Ready 实例中各个字段的处理。

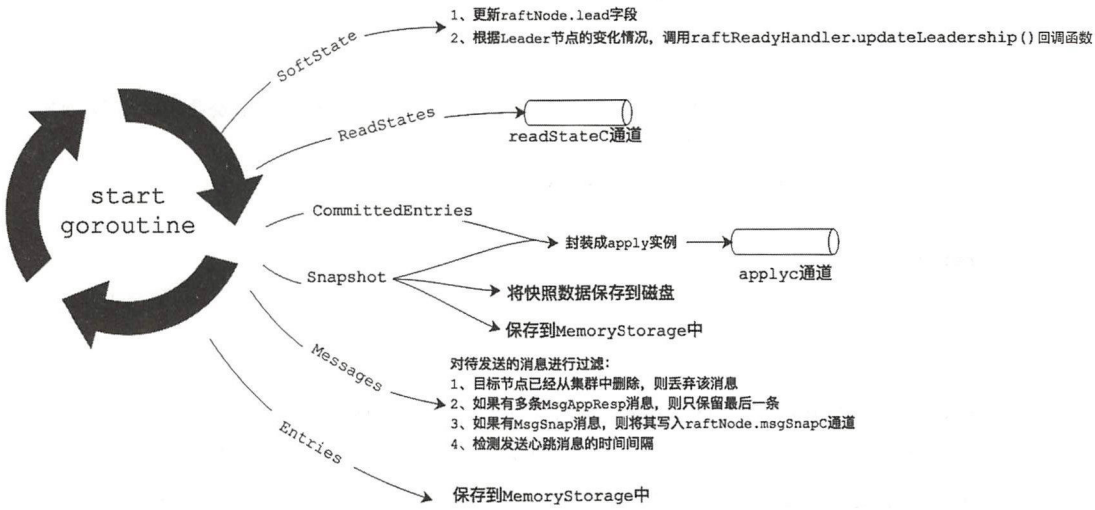


图 7-1

下面分析 raftNode.start()方法对 etcd-raft 模块返回的 Ready 实例中的 SoftState 字段，以及 ReadStates 字段的处理过程，这两个字段的含义在前面的章节中已经介绍过了，这里不再赘述，相关代码片段如下：

```
// 前面介绍过，在 SoftState 中主要封装了当前集群的 Leader 信息和当前节点角色
if rd.SoftState != nil {
    // 检测集群的 Leader 节点是否发生变化，并记录相关监控信息，这里省略了记录监控的代码
    newLeader := rd.SoftState.Lead != raft.None &&
```

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用


```
        atomic.LoadUint64(&r.lead) != rd.SoftState.Lead

    // 更新 raftNode.lead 字段，将其更新为新的 Leader 节点 ID，
    // 注意，这里读取或是更新 raftNode.lead 字段都是原子操作
    atomic.StoreUint64(&r.lead, rd.SoftState.Lead)
    islead = rd.RaftState == raft.StateLeader // 记录当前节点是否为 Leader 节点
    // 调用 raftReadyHandler 中的 updateLeadership()回调方法，其中会根据 Leader 节点是否发生变化，
    // 完成一些操作，该方法的具体实现在后面会进行详细介绍
    rh.updateLeadership(newLeader)
    r.td.Reset() // 重置全部探测器中的全部记录
}

if len(rd.ReadStates) != 0 {
    select {
    // 将 Ready.ReadStates 中的最后一项写入 readStateC 通道中
    case r.readStateC <- rd.ReadStates[len(rd.ReadStates)-1]:
    case <-time.After(internalTimeout):
        // 如果上层应用一直没有读取写入 readStateC 通道中的 ReadState 实例，会导致本次写入阻塞，
        // 这里会等待 1s，如果依然无法写入，则放弃写入并输出警告日志
        plog.Warningf("timed out sending read state")
    case <-r.stopped:
        return
    }
}
```

这里简单介绍一下 raftReadyHandler 的功能：在结构体 EtcdServer 中记录了当前节点的状态信息，例如，当前是否是 Leader 节点、Entry 记录的提交位置（committedIndex 字段）等。在 raftNode.start()方法处理 Ready 实例的过程中，会涉及这些信息的修改，raftReadyHandler 中封装了 updateLeadership 和 updateCommittedIndex 两个回调函数，这样就可以在 raftNode 中通过这两个回调函数，修改 EtcdServer 中的相应字段了。在后面会详细介绍上述两个回调函数的具体实现。

接下来看一下 raftNode.start()方法对 Ready 实例中待应用的 Entry 记录，以及快照数据的处理流程，相关的代码片段如下：

```
notifyc := make(chan struct{}, 1) // 创建 notifyc 通道
// 将 Ready 实例中的待应用 Entry 记录以及快照数据封装成 apply 实例，其中封装了 notifyc 通道，该通
// 道用来协调当前 goroutine 和 EtcdServer 启动的后台 goroutine 的执行
```

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用


```
ap := apply{
  entries:  rd.CommittedEntries, // 已提交、待应用的Entry记录
  snapshot: rd.Snapshot, // 待持久化的快照数据
  notifyc:  notifyc,
}
// 更新EtcdServer中记录的已提交位置(EtcdServer.committedIndex字段),
// EtcdServer相关的内容在后面详细介绍,这里读者暂时只需要大致了解该方法的功能即可
updateCommittedIndex(&ap, rh)

select {
case r.applyc <- ap: // 将apply实例写入applyc通道中,等待上层应用读取并进行处理
case <-r.stopped:
  return
}
```

如果当前节点处于 Leader 状态，则 raftNode.start()方法会先调用 raftNode.processMessages()方法对待发送的消息进行过滤，然后调用 rafNode.transport.Send()方法完成消息的发送，相关的代码片段如下：

```
if islead {
  r.transport.Send(r.processMessages(rd.Messages))
}
```

rafthttp.Transporter 的具体实现在前面介绍过，这里不再赘述。raftNode.processMessages()方法处理待发送消息的逻辑比较清晰，它首先会对消息进行过滤，去除目标节点已被移出集群的消息，然后分别过滤 MsgAppResp 消息、MsgSnap 消息和 MsgHeartbeat 消息，其具体实现如下：

```
func (r *raftNode) processMessages(ms []raftpb.Message) []raftpb.Message {
  sentAppResp := false
  for i := len(ms) - 1; i >= 0; i-- {  // 从后向前遍历全部待发送的消息
    if r.isIDRemoved(ms[i].To) {   // 消息的目标节点已从集群中移除,
      // 将消息的目标节点ID设置为0,通过前面对etcd-raft模块的分析可知,其发送消息的过程中,
      // 会忽略目标节点为0的消息
      ms[i].To = 0
    }

    // 只会发送最后一条MsgAppResp消息,通过前面对etcd-raft模块的分析可知,没有必要同时发送
```

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用


```
            // 多条 MsgAppResp 消息
            if ms[i].Type == raftpb.MsgAppResp {
                if sentAppResp {
                    ms[i].To = 0
                } else {
                    sentAppResp = true
                }
            }

            if ms[i].Type == raftpb.MsgSnap { // 对 MsgSnap 消息的处理
                select {
                case r.msgSnapC <- ms[i]: // 将 MsgSnap 消息写入 msgSnapC 通道中
                default:// 如果 msgSnapC 通道的缓冲区满了，则放弃此次快照的发送
                }
                ms[i].To = 0 // 将目标节点设置为 0，则 rafNode.transport 后续不会发送该消息
            }

            if ms[i].Type == raftpb.MsgHeartbeat { // 对 MsgHeartbeat 类型的消息
                // 通过前面介绍的 TimeoutDetector 进行检测，检测发往目标节点的心跳消息间隔是否过大
                ok, exceed := r.td.Observe(ms[i].To)
                if !ok {
                    // 输出警告日志，表示当前节点可能已经过载了
                }
            }
        }
        return ms
    }
```

这里简单介绍一下 TimeoutDetector 的实现，在 TimeoutDetector 中最重要的字段是 records（map[uint64]time.Time 类型），在该 map 中记录了上一次向目标节点发送心跳消息的时间（key 是节点 ID，value 是具体时间）。TimeoutDetector.Observe()方法检测两次发送心跳消息的时间间隔，其具体实现如下：

```
func (td *TimeoutDetector) Observe(which uint64) (bool, time.Duration) {
    ok := true
    now := time.Now() // 记录当前时间
    exceed := time.Duration(0)
```

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用


```
    // 从 records 中获取上次向目标节点发送心跳消息的时间
    if pt, found := td.records[which]; found {
        exceed = now.Sub(pt) - td.maxDuration // 计算两次心跳消息的时间间隔
        if exceed > 0 {
            ok = false
        }
    }
    td.records[which] = now // 更新 records 中的对应记录
    return ok, exceed
}
```

最后看一下 raftNode 对 Ready 中待持久化的 Entry 记录，以及快照数据的处理，相关代码片段如下：

```
// 通过 raftNode.storage 将 Ready 实例中携带的 HardState 信息和待持久化的 Entry 记录写入
// WAL 日志文件中，这里使用的 etcdserver.Storage 接口在前面已经介绍过了，下面会介绍
// 其具体实现，这里省略异常处理相关的代码
if err := r.storage.Save(rd.HardState, rd.Entries); err != nil ...
// 根据 HardState 信息，记录相关的监控信息(略)

if !raft.IsEmptySnap(rd.Snapshot) {
    // 通过 raftNode.storage 将 Ready 实例中携带的快照数据保存到磁盘中，错误处理(略)
    if err := r.storage.SaveSnap(rd.Snapshot); err != nil ...

    // 在后面介绍的 EtcdServer 中会启动后台 goroutine 读取前面介绍的 applyc 通道，并处理 apply 中
    // 封装快照数据。这里使用 notifyc 通道通知该后台 goroutine，该 apply 实例中的快照数据已经
    // 被持久化到磁盘，后台 goroutine 可以开始应用该快照数据了
    notifyc <- struct{}{}
    r.raftStorage.ApplySnapshot(rd.Snapshot) // 将快照数据保存到 MemoryStorage 中
}
// 将待持久化的 Entry 记录写入 MemoryStorage 中，MemoryStorage 的具体实现在前面介绍过了，这里
// 不再赘述
r.raftStorage.Append(rd.Entries)
if !islead {
    msgs := r.processMessages(rd.Messages) // 与 Leader 节点的处理逻辑类似，不再展开介绍
    // 处理 Ready 实例的过程基本结束，这里会通知 EtcdServer 启动的后台 goroutine，检测是否生成快照
```

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用


```
        notifyc <- struct{}{}
        r.transport.Send(msgs) // 发送消息
    } else {
        // 处理 Ready 实例的过程基本结束，这里会通知 EtcdServer 启动的后台 goroutine，检测是否生成快照
        notifyc <- struct{}{}
    }
```

这里我们先来看一下 Storage 接口的具体实现，etcd-server 模块提供了一个 storage 结构体，其中内嵌了前面介绍的 WAL 和 Snapshotter，其定义如下：

```
type storage struct {
    *wal.WAL
    *snap.Snapshotter
}
```

上面保存的 HardState 及待持久化 Entry 记录时调用的 storage.Save()方法，实际就是前面介绍的 WAL.Save()方法。storage 中重新实现了 SaveSnap()方法，具体实现如下：

```
func (st *storage) SaveSnap(snap raftpb.Snapshot) error {
    walsnap := walpb.Snapshot{ // 根据快照的元数据创建对应的 walpb.Snapshot 实例
        Index: snap.Metadata.Index,
        Term:  snap.Metadata.Term,
    }
    // 将 walpb.Snapshot 实例封装成 Record 记录写入 WAL 日志文件中，省略异常处理的相关代码
    err := st.WAL.SaveSnapshot(walsnap)
    // 通过 Snapshotter 将快照数据写入到磁盘，省略异常处理的相关代码
    err = st.Snapshotter.SaveSnap(snap)
    // 根据 WAL 日志文件的名称及快照的元数据，释放快照之前的 WAL 日志文件句柄
    return st.WAL.ReleaseLockTo(snap.Metadata.Index)
}
```

我们需要简单了解一下 WAL.ReleaseLockTo()方法的实现。读者先回顾一下上一章介绍的相关内容，WAL 日志的文件名中包含了该文件中第一条 Entry 记录的索引值，WAL.locks 字段中记录了当前 WAL 实例正在使用的 WAL 文件句柄。在 WAL.ReleaseLockTo()方法中，会根据 WAL 日志的文件名和快照的元数据，将比较旧的 WAL 日志文件句柄从 WAL.locks 中清除，具体实现如下：

```
func (w *WAL) ReleaseLockTo(index uint64) error {
```

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用


```
    var smaller int
    found := false

    for i, l := range w.locks {
        // 遍历 locks 字段，解析对应的 WAL 日志文件名，获取其中第一条记录的索引值
        _, lockIndex, err := parseWalName(filepath.Base(l.Name()))
        if lockIndex >= index { // 检测是否可以清除该 WAL 日志文件的句柄
            smaller = i - 1 // 真正要释放的是 smaller 之前的 WAL 日志文件
            found = true
            break
        }
    }
    // 对 smaller 进行边界检查，省略该检查的相关代码
    for i := 0; i < smaller; i++ { // 关闭 smaller 之前的全部 WAL 日志文件
        if w.locks[i] == nil {
            continue
        }
        w.locks[i].Close()
    }
    w.locks = w.locks[smaller:] // 清理 smaller 之前的文件句柄
    return nil
}
```

7.2　RaftCluster

在集群的每个节点中，都会使用 RaftCluster 记录当前集群的状态，其中各个字段的含义如下所示。

- id（types.ID 类型）：当前集群的 ID。
- token（string 类型）：当前集群的 token。
- members（map[types.ID]*Member 类型）：集群中每个节点都会有一个唯一 ID，同时对应一个 Member 实例，该 map 中记录了节点 ID 与其 Member 实例的对应关系。在 Member 中记录了对应节点暴露给集群其他节点的 URL 地址（RaftAttributes.PeerURLs）及与客户端交互的 URL 地址（Attributes.ClientURLs）。
- store（store.Store 类型）：用来持久化上述节点信息的 etcd v2 存储。
- be（backend.Backend 类型）：用来持久化上述节点信息的 etcd v3 存储。

仅供非商业用途或交流学习使用

- removed (map[types.ID]bool 类型)：从当前集群中移除的节点的 ID，在后续添加新节点时，这些 ID 不能被再次使用。

RaftCluster 中提供了一些操纵上述字段中的简单方法，这里只简单介绍其主要功能，具体实现留给读者分析，每个方法的大致功能如下所示。

- Members()方法：返回当前 RaftCluster.members 字段。
- Member()、MemberByName()方法：分别根据节点的 ID 和节点的名称查找对应的 Member 实例。
- PeerURLs()方法：将所有 Member 实例中的 PeerURLs 保存到一个集合中后返回。ClientURLs()方法类似，不再赘述。

在后面分析应用 EntryConfChange 类型记录的过程（EtcdServer.applyConfChange()方法）时会看到，其中会调用 RaftCluster 相关的方法完成新增/删除/更新节点对应的 Member，这里以 AddMember() 方法为例进行分析，剩余的 RemoveMember()、UpdateAttributes() 和 UpdateRaftAttributes()等方法留给读者自己分析。RaftCluster.AddMember()方法的具体实现如下所示。

```
func (c *RaftCluster) AddMember(m *Member) {
  if c.store != nil {
    mustSaveMemberToStore(c.store, m) // 将 Member 序列化后保存到 v2 存储中，不再展开介绍
  }
  if c.be != nil {
    mustSaveMemberToBackend(c.be, m) // 将 Member 序列化后保存到 v3 存储中，不再展开介绍
  }
  c.members[m.ID] = m // 在 RaftCluster.member 这个 map 中添加对应的 Member 实例
}
```

7.3 EtcdServer

etcdserver.Server 是 etcd 服务端的核心接口，其中定义了 etcd 服务端的主要功能，其具体定义如下：

```
type Server interface {
  Start()   // 读取配置文件，启动当前 Server 实例
  Stop()    // 关闭当前 Server 实例
  ID() types.ID // 获取当前 Server 实例的 ID
  Leader() types.ID // 获取当前集群中的 Leader 的 ID
```

仅供非商业用途或交流学习使用


```
    Do(ctx context.Context, r pb.Request) (Response, error) // 处理 Client 请求
    Process(ctx context.Context, m raftpb.Message) error // 处理 Raft 请求
    // 向当前 etcd 集群中添加一个节点
    AddMember(ctx context.Context, memb membership.Member) ([]*membership.Member, error)
    // 从当前 etcd 集群中删除一个节点
    RemoveMember(ctx context.Context, id uint64) ([]*membership.Member, error)
    // 修改集群成员属性，如果成员 ID 不存在则返回错误
    UpdateMember(ctx context.Context, updateMemb membership.Member)
        ([]*membership.Member, error)
}
```

结构体 EtcdServer 实现了 Server 接口，EtcdServer 中的字段比较多，下面介绍其中核心字段的含义，如下所示。

- Cfg（*ServerConfig 类型）：封装了配置信息，后面会详细介绍其中各个配置项。
- inflightSnapshots（int64 类型）：当前已发送出去但未收到响应的快照个数。
- appliedIndex（uint64 类型）：当前节点已应用的 Entry 记录的最大索引值。
- committedIndex（uint64 类型）：当前已提交的 Entry 记录的索引值。
- readych（chan struct{}类型）：当前节点将自身的信息推送到集群中其他节点之后，会将该通道关闭，也作为当前 EtcdServer 实例，可以对外提供服务的一个信号。
- r（raftNode 类型）：即前面介绍的 etcdserver.raftNode，它是 EtcdServer 实例与底层 etcd-raft 模块通信的桥梁。
- snapCount（uint64 类型）：当前 EtcdServer 实例每应用 snapCount 条数的 Entry 记录，就会触发一次生成快照的操作。
- w（wait.Wait 类型）：Wait 主要负责协调多个后台 goroutine 之间的执行。在 Wait 实例中维护了一个 map（map[uint64]chan interface{}类型），我们可以通过 Wait.Register(id uint64)为指定的 ID 创建一个对应的通道，ID 与通道的映射关系会记录在上述 map 中。之后，可以通过 Trigger(id uint64, x interface{})方法将参数 x 写入 id 对应的通道中，其他监听该通道的 goroutine 就可以获取该参数 x。Wait 的具体实现并不复杂，这里不再介绍其具体实现，感兴趣的读者可以参考代码进行学习。
- applyWait（wait.WaitTime 类型）：WaitTime 是在上面介绍的 Wait 之上的一层扩展。在 WaitTime 中记录的 ID 是有序的，我们可以通过 WaitTime.Wait()方法创建指定 ID 与通道之间的映射关系。WaitTime.Trigger()方法会将小于指定值的 ID 对应的通道关闭，这样就可以通知监听相应通道的 goroutine 了。WaitTime 的具体实现并不复杂，感兴趣

仅供非商业用途或交流学习使用

的读者可以参考代码进行学习。

- readwaitc（chan struct{}类型）和 readNotifier（*notifier 类型）：这两个字段主要用来协调 Linearizable Read 相关的 goroutine，后面介绍 EtcdServer.linearizableReadLoop()方法时，再详细介绍这两个字段的功能。
- stop、stopping、done（chan struct{}类型）：在 EtcdServer.start()方法中会启动多个后台 goroutine，其中一个后台 goroutine 会执行 EtcdServer.run()方法，监听 stop 通道。在 EtcdServer.Stop()方法中会将 stop 通道关闭，触发该 run goroutine 的结束。在 run goroutine 结束之前还会将关闭 stopping 和 done 通道，从而触发其他后台 goroutine 的关闭。
- id（types.ID 类型）：记录当前节点的 ID。
- attributes（membership.Attributes 类型）：记录当前节点的名称及接收集群中其他节点请求的 URL 地址。
- cluster（*membership.RaftCluster 类型）：记录当前集群中全部节点的信息。
- store（store.Store 类型）：前面介绍的 etcd v2 版本存储。
- snapshotter（*snap.Snapshotter 类型）：用来读写快照文件，前面已经详细介绍过其具体实现，这里不再赘述。
- applyV2（Applierv2 类型）：Applierv2 接口主要功能是应用 v2 版本的 Entry 记录，其底层封装了前面介绍的 v2 存储。
- applyV3、applyV3Base（applierv3 类型）：applierV3 接口主要功能是应用 v3 版本的 Entry 记录，其底层封装了前面介绍的 v3 存储。
- be（backend.Backend 类型）：v3 版本的后端存储，在前面介绍已经详细介绍过其具体实现了，这里不再赘述。
- kv（mvcc.ConsistentWatchableKV 类型）：etcd v3 版本的存储（当然，其中支持前面介绍的 watch 机制）。
- authStore（auth.AuthStore 类型）：在 backend.Backend（即这里的 be 字段）之上封装的一层存储，用于记录权限控制相关的信息，其具体实现在后面详细介绍。
- alarmStore（*alarm.AlarmStore 类型）：在 backend.Backend 之上封装的一层存储，用于记录报警相关的信息，其具体实现在后面详细介绍。
- SyncTicker（*time.Ticker 类型）：用来控制 Leader 节点定期发送 SYNC 消息的频率。
- compactor（*compactor.Periodic 类型）：Leader 节点会对存储进行定期压缩，该字段用于控制定期压缩的频率。

仅供非商业用途或交流学习使用

- reqIDGen（*idutil.Generator 类型）：用于生成请求的唯一标识。
- wg（sync.WaitGroup 类型）：在 EtcdServer.Stop()方法中会通过该字段等待所有的后台 goroutine 全部退出。
- leadElectedTime（time.Time 类型）：记录当前节点最近一次转换成 Leader 状态的时间戳。

7.3.1　初始化

在 NewServer()函数中会完成 EtcdServer 的初始化，也是 etcd 服务端生命周期的起始。该初始化的大致流程如下：

（1）定义初始化过程中使用的变量，创建当前节点使用的目录，相关代码如下所示。

```
var (
   w  *wal.WAL // 用于管理 WAL 日志文件的 WAL 实例
   n  raft.Node // etcd-raft 模块中的 Node 实例
   s  *raft.MemoryStorage // 前面介绍的 MemoryStorage 实例
   id types.ID // 记录了当前节点的 ID
   cl *membership.RaftCluster // 当前集群中的所有成员的信息
   remotes  []*membership.Member
   snapshot *raftpb.Snapshot
)
// 每个 etcd 节点都有会将其数据保存到“节点名称.etcd/member”目录下。如果在下面没有特殊说明，
// 则提到的目录都是该目录下的子目录。这里会先检测该目录是否存在，如果不存在就创建该目录，错误处理(略)
if terr := fileutil.TouchDirAll(cfg.DataDir); ...
// 确定 snap 目录是否存在，该目录是用来存放快照文件的，错误处理(略)
if err = fileutil.TouchDirAll(cfg.SnapDir()); ...
```

（2）根据配置项初始化 etcd-raft 模块使用到的相关组件，例如，检测当前 wal 目录下是否存在 WAL 日志文件、初始化 v2 存储、查找 BoltDB 数据库文件、创建 Backend 实例、创建 RoundTripper 实例等。相关的代码片段如下所示。

```
// 检测 wal 目录下是否存在 WAL 日志文件
haveWAL := wal.Exist(cfg.WALDir())
// 创建 v2 版本存储，这里指定指定了 StoreClusterPrefix 和 StoreKeysPrefix 两个只读目录，
// 这两个常量值分别是“/0”和“/1”
st := store.New(StoreClusterPrefix, StoreKeysPrefix)
```

仅供非商业用途或交流学习使用


```
// 创建 Snapshotter 实例，用来读写 snap 目录下的快照文件
ss := snap.New(cfg.SnapDir())
bepath := cfg.backendPath() // 获取 BoltDB 数据库文件存放的路径
beExist := fileutil.Exist(bepath) // 检测 BoltDB 数据库文件是否存在
// 创建 Backend 实例，其中会单独启动一个后台 goroutine 来创建 Backend 实例，
// Backend 的原理及具体实现在前面已经介绍过，这里不再赘述
be := openBackend(cfg)
// 根据配置创建 RoundTripper 实例，RoundTripper 主要负责实现网络请求等功能，其具体实现
// 在前面详细介绍过了，这里不再赘述
prt, err := rafthttp.NewRoundTripper(cfg.PeerTLSInfo, cfg.peerDialTimeout())
```

这里需要简单介绍一下 openBackend()函数，在该函数中会启动一个后台 goroutine 完成 Backend 实例的初始化，具体实现如下：

```
func openBackend(cfg *ServerConfig) backend.Backend {
    fn := cfg.backendPath()
    beOpened := make(chan backend.Backend)
    go func() { // 启动一个后台 goroutine 并将新建的 Backend 实例写入 beOpened 通道中
        beOpened <- newBackend(cfg) // 新建 Backend 实例
    }()
    select {
    case be := <-beOpened:
        return be
    case <-time.After(time.Second): // 初始化 Backend 实例超时，会输出日志
    }
    return <-beOpened // 阻塞等待 Backend 实例初始化完成
}
```

（3）根据前面对 WAL 日志文件的查找结果及当前节点启动时的配置信息，初始化 etcd-raft 模块中的 Node 实例，大致分为如下三种场景。

```
switch {
    case !haveWAL && !cfg.NewCluster: // 没有 WAL 日志文件且当前节点正在加入一个正在运行的集群
        ... ...
    case !haveWAL && cfg.NewCluster: // 没有 WAL 日志文件且当前集群是新建的
        ... ...
    case haveWAL:// 存在 WAL 日志文件
        ... ...
    default: // 不支持其他场景
```

仅供非商业用途或交流学习使用


```
        return nil, fmt.Errorf("unsupported bootstrap config")
    }
```

场景一：首先分析第一种场景，即当前节点的 wal 目录下不存在 WAL 日志文件，当前节点正在加入一个正在运行的集群。该场景的相关代码如下所示。

```
// 对配置的合法性进行检查，其中涉及配置信息中是否包含当前节点的相关信息，以及集群各个节点暴
// 露的 URL 地址是否重复等。VerifyJoinExisting()方法比较简单，这里不展开介绍，感兴趣
// 的读者请参考源码。
if err = cfg.VerifyJoinExisting();...

// 根据配置信息，创建 RaftCluster 实例和其中的 Member 实例，在后面具体展开介绍创建过程，
// 错误处理(略)
cl, err = membership.NewClusterFromURLsMap(cfg.InitialClusterToken,
            cfg.InitialPeerURLsMap)

// getRemotePeerURLs()函数会过滤当前节点的信息，然后排序集群中其他节点暴露的 URL 地址并返回
// GetClusterFromRemotePeers 从集群中其他节点请求集群信息并创建相应的 RaftCluster 实例，
// 然后将其与 getRemotePeerURLs()返回值比较，错误处理(略)
existingCluster, gerr := GetClusterFromRemotePeers(getRemotePeerURLs(cl,
     cfg.Name), prt)
if err = membership.ValidateClusterAndAssignIDs(cl, existingCluster);...

// 检测正在运行的集群与当前节点的版本，保证其版本相互兼容(略)
cl.SetID(existingCluster.ID()) // 更新本地 RaftCluster 实例中的集群 ID
cl.SetStore(st) // 设置本地 RaftCluster 实例中的 store 字段
// 设置本地 RaftCluster 实例中的 be 字段，同时会在 BoltDB 中初始化后续使用到
// 的 Bucket(如："members"、"members_removed"、"cluster" 三个 Bucket)
cl.SetBackend(be)
// 调用 startNode()函数，初始化前面介绍的 raft.Node 实例及相关组件
id, n, s, w = startNode(cfg, cl, nil)
```

先介绍 NewClusterFromURLsMap()函数，在 ServerConfig.InitialPeerURLsMap 中封装了集群中每个节点的名称与其提供的 URL 之间的映射，在 NewClusterFromURLsMap()函数中会根据该映射关系创建相应的 Member 实例和 RaftCluster 实例，其具体实现如下所示。

```
func NewClusterFromURLsMap(token string, urlsmap types.URLsMap)(*RaftCluster, error) {
```

仅供非商业用途或交流学习使用


```
    c := NewCluster(token) // 创建 RaftCluster 实例
    for name, urls := range urlsmap { // 遍历集群中节点的地址，并创建对应的 Member
        m := NewMember(name, urls, token, nil)
        if _, ok := c.members[m.ID]; ...
        c.members[m.ID] = m
    }
    c.genID() // 通过所有节点的 ID，为集群生成一个 ID
    return c, nil
}
```

getRemotePeerURLs()函数会过滤掉当前节点提供的 URL 地址，然后排序集群中其他节点暴露的 URL 地址并返回，其具体实现如下：

```
func getRemotePeerURLs(cl *membership.RaftCluster, local string) []string {
    us := make([]string, 0)
    for _, m := range cl.Members() {
        if m.Name == local { // 过滤本地节点
            continue
        }
        us = append(us, m.PeerURLs...)
    }
    sort.Strings(us) // 对其他节点的 URL 地址进行排序
    return us
}
```

GetClusterFromRemotePeers()函数底层通过调用 getClusterFromRemotePeers()函数实现，其中会从集群中其他节点请求当前集群的信息，然后封装成 RaftCluster 实例返回，具体实现如下：

```
func getClusterFromRemotePeers(urls []string, timeout time.Duration,
     logerr bool, rt http.RoundTripper) (*membership.RaftCluster, error) {
    cc := &http.Client{ // 创建一个 HTTP 客户端
        Transport: rt,
        Timeout:   timeout,
    }
    for _, u := range urls { // 请求集群中其他节点，获取集群信息，如果发生错误，则换一个节点请求
        // 请求集群中其他节点提供的 HTTP 接口(members)，错误处理(略)
        resp, err := cc.Get(u + "/members")
        // 读取响应，响应内容是 JSON 格式，错误处理(略)
```

```
        b, err := ioutil.ReadAll(resp.Body)
        resp.Body.Close()
        var membs []*membership.Member
        // 解析 JSON 得到 Member 实例数组，错误处理(略)
        if err = json.Unmarshal(b, &membs); ...
        // 从响应头中获取集群 ID，错误处理(略)
        id, err := types.IDFromString(resp.Header.Get("X-Etcd-Cluster-ID"))
        if len(membs) > 0 { // 创建 RaftCluster 实例并返回
            return membership.NewClusterFromMembers("", id, membs), nil
        }
    }
}
```

在从远端获取到 RaftCluster 实例之后会调用 ValidateClusterAndAssignIDs()函数，将其与本地生成的 RaftCluster 实例进行比较，具体的比较过程如下：

```
func ValidateClusterAndAssignIDs(local *RaftCluster, existing *RaftCluster) error {
    // 获取本地 RaftCluster 实例和远端 RaftCluster 实例中记录 Member 实例，需要注意的是，
    // 这里返回的是 RaftCluster.memebers 字段中记录的 Member 实例的副本
    ems := existing.Members()
    lms := local.Members()
    // 首先，比较 ems 和 lms 的长度是否相同，如果不相同，则直接返回错误(略)
    sort.Sort(MembersByPeerURLs(ems))    // 对 ems 和 lms 进行排序
    sort.Sort(MembersByPeerURLs(lms))

    for i := range ems { // 遍历 ems，检测 lms 中对应节点暴露的 URL 地址是否匹配
        if !netutil.URLStringsEqual(ctx, ems[i].PeerURLs, lms[i].PeerURLs) {
            return fmt.Errorf("unmatched member while checking PeerURLs")
        }
        lms[i].ID = ems[i].ID // 记录匹配的 Member 实例
    }
    local.members = make(map[types.ID]*Member) // 清空本地 RaftCluster.members 字段
    for _, m := range lms {
        local.members[m.ID] = m // 更新本地 RaftCluster.members 字段
    }
    return nil
}
```

完成上述检测之后，会设置更新本地 RaftCluster 实例的相关字段。最后，调用 startNode() 函数初始化前面介绍的 etcd-raft 模块中的 Node 实例，startNode()方法的具体实现如下：

```
func startNode(cfg *ServerConfig, cl *membership.RaftCluster, ids []types.ID)
    (id types.ID, n raft.Node, s *raft.MemoryStorage, w *wal.WAL) {
  var err error
  // 根据当前节点的名称，在 RaftCluster 中查找对应的 Member 实例
  member := cl.MemberByName(cfg.Name)
  metadata := pbutil.MustMarshal( // 将节点的 id 和当前集群的 id 封装后进行序列化
    &pb.Metadata{
      NodeID:    uint64(member.ID), // 节点的 id
      ClusterID: uint64(cl.ID()), // 集群的 id
    },
  )
  // 创建 WAL 日志文件，并将上述元数据信息作为第一条日志记录写入 WAL 日志文件中，错误处理(略)
  if w, err = wal.Create(cfg.WALDir(), metadata);...

  // 为集群中每个节点，创建对应的 Peer 实例
  peers := make([]raft.Peer, len(ids))
  for i, id := range ids {
    ctx, err := json.Marshal((*cl).Member(id)) // 直接将 Member 实例序列化
    // 记录了节点 id 和 Member 实例序列化后的数据
    peers[i] = raft.Peer{ID: uint64(id), Context: ctx}
  }
  id = member.ID // 当前节点的 ID
  s = raft.NewMemoryStorage() // 新建 MemoryStorage 实例
  c := &raft.Config{ // 初始化 Node 实例时使用的相关配置
    ID:            uint64(id),
    ElectionTick:   cfg.ElectionTicks,
    HeartbeatTick:  1,
    Storage:        s,
    MaxSizePerMsg:  maxSizePerMsg,
    MaxInflightMsgs: maxInflightMsgs,
    CheckQuorum:    true,
  }
  // 创建 raft.Node，raft.Node 的相关内容在前面已经详细介绍过了，这里不再展开介绍
  n = raft.StartNode(c, peers)
```

```
    // 快速推进选举计时器，只留下一个逻辑时间的刻度(即一次心跳时间)，这样在集群启动的时候，
    // 可以迅速进行 Leader 节点的选举
    advanceTicksForElection(n, c.ElectionTick)
    return
}
```

场景二：紧接着分析第二个场景的处理过程，即当前节点的 wal 目录下不存在 WAL 日志文件且当前集群是新建的，相关代码如下所示。

```
if err = cfg.VerifyBootstrap();...// 对当前节点启动使用的配置进行一系列检测
// 根据配置信息创建本地 RaftCluster 实例，NewClusterFromURLsMap()函数在前面已经介绍
// 过了，这里不再赘述
cl, err = membership.NewClusterFromURLsMap(cfg.InitialClusterToken,
     cfg.InitialPeerURLsMap)
// 根据当前节点的名称，从 RaftCluster 中查找当前节点对应的 Member 实例
m := cl.MemberByName(cfg.Name)

// 从集群中其他节点获取当前集群的信息，检测是否有同名的节点已经启动了
if isMemberBootstrapped(cl, cfg.Name, prt, cfg.bootstrapTimeout()) {
   return nil, fmt.Errorf("member %s has already been bootstrapped", m.ID)
}
// 根据当前的配置检测是否要使用 Discover 模式启动，这个模式在后面会进行介绍(略)

cl.SetStore(st) // 设置本地 RaftCluster 实例中的 store 字段
// 设置本地 RaftCluster 实例中的 be 字段，同时会在 BoltDB 中初始化后续使用的 Bucket
cl.SetBackend(be)
// 调用 startNode()函数，初始化前面介绍的 raft.Node 实例及相关组件，不再展开详述
id, n, s, w = startNode(cfg, cl, cl.MemberIDs())
```

这里简单介绍 ServerConfig.VerifyBootstrap()方法，其中主要检测当前配置中是否包含当前节点，检测当前节点提供的 URL 与 initial-advertise-peer-urls 配置项是否相同，检测集群配置中是否存在重复的 URL 地址，等等，具体实现比较简单，感兴趣的读者可以参考其代码。

另一个需要简单介绍的是 isMemberBootstrapped()函数，该函数会请求远端节点，然后检测当前集群中是否已存在相同的节点，具体实现如下：

```
func isMemberBootstrapped(cl *membership.RaftCluster, member string,
        rt http.RoundTripper, timeout time.Duration) bool {
   // 从远端节点获取集群信息，并返回 RaftCluster 实例，该方法在前面已经详细分析过了
```

```
    rcl, err := getClusterFromRemotePeers(getRemotePeerURLs(cl, member),
        timeout, false, rt)
    id := cl.MemberByName(member).ID // 当前节点的 id
    m := rcl.Member(id) // 根据 id 获取对应的 Member 实例
    if m == nil {
        return false
    }
    if len(m.ClientURLs) > 0 { // 检测已启动的相同节点是否对 Client 暴露了 URL
        return true
    }
    return false
}
```

场景三：最后一个场景就是 wal 目录下存在 WAL 日志文件的场景，其相关的代码片段如下所示。

```
// 检测 member 和 wal 文件夹都是可写的(略)
snapshot, err = ss.Load() // 加载快照数据，错误处理(略)
if snapshot != nil { // 根据加载的快照数据，对 v2 存储和 v3 存储进行恢复
    // 使用快照数据恢复 v2 存储，具体恢复过程在前面已经介绍过了，这里不再赘述，错误处理(略)
    if err = st.Recovery(snapshot.Data); ...
    // 使用快照数据恢复 v3 存储，错误处理(略)
    if be, err = recoverSnapshotBackend(cfg, be, *snapshot); ...
}
if !cfg.ForceNewCluster {
    // 重启 raft.Node 节点
    id, cl, n, s, w = restartNode(cfg, snapshot)
} else { // 启动单节点的方式与上面的 restartNode()函数类似，留给读者自己分析
    id, cl, n, s, w = restartAsStandaloneNode(cfg, snapshot)
}
cl.SetStore(st) // 设置本地 RaftCluster 实例的相关字段，与前面两种场景类似
cl.SetBackend(be)
// 从 v2 存储中，恢复集群中其他节点的信息，其中还会检测当前服务端的版本与 WAL 日志文件以及快
// 照数据的版本之间的兼容性
cl.Recover(api.UpdateCapability)
```

该场景主要是通过快照数据恢复当前节点的 v2 和 v3 存储，然后恢复 etcd-raft 模块中的 Node 实例，其中 v2 存储的恢复比较简单，就是加载快照文件中的 JSON 数据，这里不再展开介绍。

重点来看 v3 存储的恢复，该过程是在 recoverSnapshotBackend()函数中实现的，其中会检测前面创建的 Backend 实例是否可用（即包含了快照数据所包含的全部 Entry 记录），如果可用则继续使用该 Backend 实例，如果不可用则根据快照的元数据查找可用的 BoltDB 数据库文件，并创建新的 Backend 实例。其中查找可用的 BoltDB 数据库文件和新建 Backend 实例等操作是在 openSnapshotBackend()函数中完成的，整个过程的具体实现如下：

```
func recoverSnapshotBackend(cfg *ServerConfig, oldbe backend.Backend,
       snapshot raftpb.Snapshot) (backend.Backend, error) {
  var cIndex consistentIndex
  // 省略获取 cInde 值的相关代码
  if snapshot.Metadata.Index <= kv.ConsistentIndex() {
    return oldbe, nil    // 若当前使用的 Backend 实例可用，则直接返回
  }
  oldbe.Close() // 旧 Backend 实例不可用，则关闭旧的 Backend 实例，并重建新的 Backend 实例
  return openSnapshotBackend(cfg, snap.New(cfg.SnapDir()), snapshot)
}

// 重建新的 Backend 实例的 openSnapshotBackend()函数
func openSnapshotBackend(cfg *ServerConfig, ss *snap.Snapshotter,
       snapshot raftpb.Snapshot) (backend.Backend, error) {
  // 根据快照元数据查找对应的 BoltDB 数据库文件，在后面介绍快照消息的处理时，会介绍
  // BoltDB 数据库文件的命名和编号
  snapPath, err := ss.DBFilePath(snapshot.Metadata.Index)
  // 将可用的 BoltDB 数据库文件移动到指定的目录中
  if err := os.Rename(snapPath, cfg.backendPath());...
  return openBackend(cfg), nil // 新建 Backend 实例
}
```

restartNode()函数会根据配置信息和加载的快照数据，重启 etcd-raft 模块的 Node 实例，其具体实现如下：

```
func restartNode(cfg *ServerConfig, snapshot *raftpb.Snapshot)
   (types.ID, *membership.RaftCluster, raft.Node, *raft.MemoryStorage, *wal.WAL) {
  var walsnap walpb.Snapshot
  if snapshot != nil {
    walsnap.Index, walsnap.Term = snapshot.Metadata.Index, snapshot.Metadata.Term
  }
  // 根据快照的元数据，查找合适的 WAL 日志并完成 WAL 日志的回放
```

```
    w, id, cid, st, ents := readWAL(cfg.WALDir(), walsnap)
    cl := membership.NewCluster("")
    cl.SetID(cid)
    s := raft.NewMemoryStorage() // 创建 MemoryStorage 实例
    if snapshot != nil {
        s.ApplySnapshot(*snapshot) // 将快照数据记录到 MemoryStorage 实例中
    }
    s.SetHardState(st) // 根据 WAL 日志文件回放的结果设置 HardState
    s.Append(ents) // 向 MemoryStorage 实例中追加快照数据之后的 Entry 记录
    c := &raft.Config{
        ID:              uint64(id),
        ElectionTick:    cfg.ElectionTicks,
        HeartbeatTick:   1,
        Storage:         s,
        MaxSizePerMsg:   maxSizePerMsg,
        MaxInflightMsgs: maxInflightMsgs,
        CheckQuorum:     true,
    }

    n := raft.RestartNode(c) // 根据上面的 raft.Config，重建 raft.Node 实例
    // 快速推进选举计时器，在集群启动的时候，可以迅速进行 Leader 节点的选举，该函数在前面介绍过了，
    // 这里不再赘述
    advanceTicksForElection(n, c.ElectionTick)
    return id, cl, n, s, w
}
```

（4）创建 EtcdServer 实例，并初始化其各个字段，具体实现如下所示。

```
heartbeat := time.Duration(cfg.TickMs) * time.Millisecond
    srv = &EtcdServer{ // 创建 EtcdServer 实例
        readych:     make(chan struct{}),
        Cfg:         cfg,
        snapCount:   cfg.SnapCount,
        errorc:      make(chan error, 1),
        store:       st,
        snapshotter: ss,
        r: *newRaftNode( // 使用前面创建的各个组件创建 etcdserver.raftNode 实例
            raftNodeConfig{
```

```
            isIDRemoved: func(id uint64) bool { return cl.IsIDRemoved(types.ID(id)) },
            Node:        n,
            heartbeat:   heartbeat,
            raftStorage: s,
            storage:     NewStorage(w, ss),
        },
    ),
    id:          id,
    attributes:    membership.Attributes{Name: cfg.Name,
        ClientURLs: cfg.ClientURLs.StringSlice()},
    cluster:       cl,
    SyncTicker:    time.NewTicker(500 * time.Millisecond),
    peerRt:        prt,
    reqIDGen:      idutil.NewGenerator(uint16(id), time.Now()),
}
// 初始化 EtcdServer.applyV2 字段
srv.applyV2 = &applierV2store{store: srv.store, cluster: srv.cluster}
srv.be = be // 初始化 EtcdServer.be 字段
minTTL := time.Duration((3*cfg.ElectionTicks)/2) * heartbeat
// 因为在 store.restore()方法中除了恢复内存索引，还会重新绑定键值对与对应的 Lease，
// 所以需要先恢复 EtcdServer.lessor，再恢复 EtcdServer.kv 字段
srv.lessor = lease.NewLessor(srv.be, int64(math.Ceil(minTTL.Seconds())))
srv.kv = mvcc.New(srv.be, srv.lessor, &srv.consistIndex)
// 根据 ConsistentIndex 检测当前的 Backend 实例是否可用(略)
srv.authStore = auth.NewAuthStore(srv.be, tp)   // 初始化 EtcdServer.authStore 字段
if h := cfg.AutoCompactionRetention; h != 0 {  // 启动后台 goroutine，进行自动压缩
    srv.compactor = compactor.NewPeriodic(h, srv.kv, srv)
    srv.compactor.Run()
}
srv.applyV3Base = &applierV3backend{srv} // 初始化 applyV3Base 字段
if err = srv.restoreAlarms();...// 初始化 alarmStore 字段以及 applyV3 字段
// 创建 rafthttp.Transport 实例，其具体原理在前面已经介绍过了，这里不再赘述
tr := &rafthttp.Transport{
  ... ...
  // 需要注意，在这里传入的 rafthttp.Raft 接口实现是 EtcdServer 实例。EtcdServer
  // 对 Raft 接口的实现比较简单，它会直接将调用委托给底层的 raftNode 实例。对于 EtcdServer
  // 实现的这些 Raft 接口方法不再详细描述，感兴趣的读者可以参考代码进行分析
  Raft:  srv,
```

```
    ... ...
  }
  // 启动 rafthttp.Transport 实例
  if err = tr.Start(); ...
  // 向 rafthttp.Transport 实例中添加集群中各个节点对应的 Peer 实例和 Remote 实例
  for _, m := range remotes {
    if m.ID != id {
      tr.AddRemote(m.ID, m.PeerURLs)
    }
  }
  for _, m := range cl.Members() {
    if m.ID != id {
      tr.AddPeer(m.ID, m.PeerURLs)
    }
  }
  srv.r.transport = tr // 设置 raft.Node.transport 字段
```

到此为止，EtcdServer 实例的初始化流程就介绍完了。

7.3.2 注册Handler

在执行完 EtcdServer.NewServer()方法之后，Transport 实例已经启动，我们可以在其上注册多个 Handler 实例，这些 Handler 实例主要用于集群内部各节点之间的通信。该过程在 NewPeerHandler()函数中完成，其具体实现如下所示。

```
func NewPeerHandler(s *etcdserver.EtcdServer) http.Handler {
  var lh http.Handler
  l := s.Lessor()
  if l != nil { // 初始化处理 Lease 的 Handler 实例
    lh = leasehttp.NewHandler(l, func() <-chan struct{} { return s.ApplyWait() })
  }
  return newPeerHandler(s.Cluster(), s.RaftHandler(), lh)
}
// newPeerHandler()方法的具体实现如下
func newPeerHandler(cluster api.Cluster, raftHandler http.Handler,
      leaseHandler http.Handler) http.Handler {
  // 创建peerMemberHandler实例,在前面初始化EtcdServer实例时会从远端节点请求当前集群的信息,
```

```
    // 而远端节点就是通过该 Handler 对此请求进行响应的
    mh := &peerMembersHandler{
        cluster: cluster,
    }
    mux := http.NewServeMux()
    mux.HandleFunc("/", http.NotFound) // 使用默认的 Handler，直接返回 404 状态码
    // 注册 Transport.Handler()方法返回的 Handler
    mux.Handle(rafthttp.RaftPrefix, raftHandler)
    mux.Handle(rafthttp.RaftPrefix+"/", raftHandler)
    mux.Handle(peerMembersPrefix, mh) // 将上述 peerMemberHandler 实例注册到“/members”路径上
    // 注册 lease 相关的 Handler 实例，lease 相关的内容后面介绍
    if leaseHandler != nil {
        mux.Handle(leasehttp.LeasePrefix, leaseHandler)
        mux.Handle(leasehttp.LeaseInternalPrefix, leaseHandler)
    }
    // 注册 versionHandler，用于返回当前节点的版本信息
    mux.HandleFunc(versionPath, versionHandler(cluster, serveVersion))
    return mux
}
```

当前节点可以向集群中其他节点提供多个 URL 地址用于交互，当前节点会为每个对外使用的 URL 地址创建一个 peerListener 实例，其中封装了上述注册的 Handler，以及启动和关闭 HTTP 服务的回调函数，相关代码片段如下：

```
ph := etcdhttp.NewPeerHandler(e.Server) // 注册 Handler 实例，上面已介绍过其实现，这里不再
                                        // 赘述
for _, p := range e.Peers {
    srv := &http.Server{ // 为每个 URL 地址创建一个 http.Server
        Handler:     ph,
        ReadTimeout: 5 * time.Minute,
    }
    l := p.Listener  // 初始化 peerListener 实例
    // 设置启动 Http 服务端的回调函数，并未真正启动
    p.serve = func() error { return srv.Serve(l) }
    p.close = func(ctx context.Context) error { // 设置关闭 HTTP 服务端的回调函数
        return srv.Shutdown(ctx)
    }
}
```

7.3.3 启动

在执行完 EtcdServer.NewServer()、注册 Handler 等初始化操作之后，下面会紧接着调用 EtcdServer.Start()方法启动当前节点，对外提供服务。EtcdServer.Start()方法的具体实现如下所示。

```
func (s *EtcdServer) Start() {
  // 其中会启动一个后台 goroutine，执行 EtcdServer.run()方法
  s.start()
  // 启动一个后台 goroutine，将当前节点的相关信息发送到集群其他节点
  s.goAttach(func() { s.publish(s.Cfg.ReqTimeout()) })
  // 启动一个后台 goroutine，定义清理 WAL 日志文件和快照文件
  s.goAttach(s.purgeFile)
  // 启动一个后台 goroutine，实现一些监控相关的功能，这里不展开详述，对监控感兴趣的读者可以参
  // 考源码进行学习
  s.goAttach(func() { monitorFileDescriptor(s.stopping) })
  // 启动一个后台 goroutine 监控集群中其他节点的版本信息，主要是在版本升级的时候使用，这里不展开
  // 详述，对感兴趣的读者可以参考源码进行学习
  s.goAttach(s.monitorVersions)
  // 启动一个后台 goroutine，用来实现 Linearizable Read 的功能
  s.goAttach(s.linearizableReadLoop)
}
```

1. start

下面先简单介绍一下 EtcdServer.start()方法，该方法会初始化 EtcdServer 实例中剩余的未初始化字段，然后启动后台 goroutine 来执行 EtcdServer.run()方法（为了后面便于描述，将该 goroutine 称为“run goroutine”）。run()方法是 EtcdServer 启动的核心，其中会启动前面介绍的 etcdserver.raftNode 实例，然后处理 etcd-raft 模块返回的 Ready 实例，EtcdServer.run()方法的具体实现如下（由于篇幅限制，省略了加锁解锁和错误处理的相关代码）：

```
func (s *EtcdServer) run() {
  sn, err := s.r.raftStorage.Snapshot()
  sched := schedule.NewFIFOScheduler()// FIFO 调度器
  var syncC <-chan time.Time
  // setSyncC()和 getSync()方法是用来设置发送 SYNC 消息的定时器，这两个回调函数会
  // 与下面介绍的 raftReadyHandler 配合使用
  setSyncC := func(ch <-chan time.Time) {
    syncC = ch
```

```
}
getSyncC := func() (ch <-chan time.Time) {
    ch = syncC
}

rh := &raftReadyHandler{
    // raftNode在处理etcd-raft模块返回的Ready.SoftState字段时，会调用
    // raftReadyHandler.updateLeadership()回调函数，其中会根据当前节点的状态和
    // Leader节点是否发生变化完成一些相应的操作
    updateLeadership: func(newLeader bool) {
        if !s.isLeader() {
            if s.lessor != nil {
                s.lessor.Demote()// 调用lessor.Demote()将该节点的lessor实例降级
            }
            if s.compactor != nil { // 非Leader节点暂停自动压缩，自动压缩的功能在后面详细介绍
                s.compactor.Pause()
            }
            setSyncC(nil) // 非Leader节点不会发送SYNC消息，将该定时器设置为nil
        } else {
            if newLeader {
                t := time.Now()
                //  如果发生Leader节点的切换，且当前节点成为Leader节点，则初始化
                // leadElectedTime字段，该字段记录了当前节点最近一次成为Leader节点的时间
                s.leadElectedTime = t
            }
            setSyncC(s.SyncTicker.C) // Leader节点会定期发送SYNC消息，恢复该定时器
            if s.compactor != nil {  // 重启自动压缩的功能
                s.compactor.Resume()
            }
        }
    },
    // 在raftNode处理apply实例时会调用updateCommittedIndex()函数，该函数会根据apply
    // 实例中封装的待应用Entry记录和快照数据确定当前的committedIndex值，然后
    // 调用raftReadyHandler中的同名回调函数更新EtcdServer.committedIndex字段值
    updateCommittedIndex: func(ci uint64) {
        cci := s.getCommittedIndex()
        if ci > cci {
            s.setCommittedIndex(ci)
```

```
            }
        },
    }
    // 启动 raftNode，其中会启动后台 goroutine 处理 etcd-raft 模块返回的 Ready 实例，前面已经介
    // 绍过，不再赘述
    s.r.start(rh)

    ep := etcdProgress{ // 记录当前快照相关的元数据信息和已应用 Entry 记录的位置信息
        confState: sn.Metadata.ConfState,
        snapi:    sn.Metadata.Index,
        appliedt: sn.Metadata.Term,
        appliedi: sn.Metadata.Index,
    }

    for {
        select {
        case ap := <-s.r.apply(): // 读取 raftNode.applyc 通道中的 apply 实例并进行处理
            f := func(context.Context) { s.applyAll(&ep, &ap) }
            sched.Schedule(f)
        case <-getSyncC(): // 定时发送 SYNC 消息
            if s.store.HasTTLKeys() { // 如果 v2 存储中只有永久节点，则无须发送 SYNC
                s.sync(s.Cfg.ReqTimeout())// 发送 SYNC 消息的目的是为了清理 v2 存储中的过期节点
            }
        // 省略 lease 相关代码，其相关实现后面介绍
            }
    }
}
```

在开始深入介绍 EtcdServer.applyAll()方法对 apply 实例的处理之前，先来分析 EtcdServer.Start()方法中启动的后台 goroutine 都完成了哪些操作。在 EtcdServer.goAttach()方法中会启动一个后台 goroutine 执行传入的函数，其具体实现如下所示。

```
func (s *EtcdServer) goAttach(f func()) {
    select {
    case <-s.stopping:  // 检测当前 EtcdServer 实例是否已停止
        return
    default:
    }
```

```
    s.wg.Add(1) // 调用 EtcdServer.Stop()时，需要等待该后台 goroutine 结束之后才返回
    go func() { // 启动一个后台 goroutine，执行传入的回调函数
        defer s.wg.Done() // 该后台 goroutine 结束时调用
        f()
    }()
}
```

2. publish

在 EtcdServer.Start()方法中会启动一个后台 goroutine 将当前节点的相关信息发送到集群其他节点（即将当前节点注册到集群当中），相应的逻辑位于 EtcdServer.publish()方法中，其具体实现如下：

```
func (s *EtcdServer) publish(timeout time.Duration) {
    // 将 EtcdServer.attributes 字段序列化成 JSON 格式，错误处理(略)
    b, err := json.Marshal(s.attributes)
    req := pb.Request{ // 将上述 JSON 数据封装成 PUT 请求
        Method: "PUT",
        Path:   membership.MemberAttributesStorePath(s.id),
        Val:    string(b),
    }

    for {
        ctx, cancel := context.WithTimeout(s.ctx, timeout)
        _, err := s.Do(ctx, req) // 调用 EtcdServer.DO()方法处理该请求
        cancel()
        switch err {
        case nil:
            // 将当前节点信息发送到集群其他节点之后，会将 readych 通道关闭，从而实现通知其他 goroutine
            // 的目的，后面还会介绍监听的 readych 通道的 goroutine
            close(s.readych)
            return
        // 如果出现错误，则会输出相应的日志，然后继续当前的 for 循环，直至注册成功(略)
        }
    }
}
```

在 EtcdServer.Do()方法中会根据请求的 Method 字段值进行分类处理，其大致实现如下：

```
func (s *EtcdServer) Do(ctx context.Context, r pb.Request) (Response, error) {
  r.ID = s.reqIDGen.Next() // 为请求生成唯一 ID
  if r.Method == "GET" && r.Quorum {
    r.Method = "QGET"
  }
  // EtcdServer 实现了 v2API 接口，这里将当前的 EtcdServer 实例转换成 v2API
  v2api := (v2API)(&v2apiStore{s})
  switch r.Method { // 下面会根据请求的 Method 字段调用对应的方法
  case "POST":
    return v2api.Post(ctx, &r)
  case "PUT":
    return v2api.Put(ctx, &r)
  case "DELETE":
    return v2api.Delete(ctx, &r)
  case "QGET":
    return v2api.QGet(ctx, &r)
  case "GET":
    return v2api.Get(ctx, &r)
  case "HEAD":
    return v2api.Head(ctx, &r)
  }
  return Response{}, ErrUnknownMethod
}
```

在 EtcdServer 实现的 v2API 接口的方法中，除了 Get()方法和 Head()方法，其他方法都是通过调用 processRaftRequest()方法实现的。在 processRaftRequest()方法中会将请求交给 etcd-raft 模块进行处理，并且阻塞等待请求处理结束，具体实现如下：

```
func (a *v2apiStore) processRaftRequest(ctx context.Context, r *pb.Request)
     (Response, error) {
  data, err := r.Marshal()  // 将请求进行序列化
  // 为该请求注册一个相应的通道，在 Wait 中维护该请求 id 与该通道的对应关系，Wait 的功能在前面已经
  // 介绍过了，这里不再赘述
  ch := a.s.w.Register(r.ID)
  a.s.r.Propose(ctx, data) // 将请求交给底层的 etcd-raft 模块进行处理
  select {
  // 阻塞等待，待该请求处理完成之后，从请求对应的通道中获取对应响应，稍后在介绍 EtcdServer.apply*()
  // 方法时会看到其中为相应的请求调用了 Wait.Trigger()方法
```

```
    case x := <-ch:
      resp := x.(Response)
      return resp, resp.err
    // 省略其他出现错误的场景
    }
    return Response{}, ErrStopped
}
```

3. purgeFile

在 EtcdServer.Start()方法中会启动两个后台 goroutine，其中一个后台 goroutine 负责定期清理 WAL 日志文件，另一个后台 goroutine 负责定期清理快照文件，相应的逻辑位于 EtcdServer.purgeFile()方法中，具体实现如下：

```
func (s *EtcdServer) purgeFile() {
  var serrc, werrc <-chan error
  if s.Cfg.MaxSnapFiles > 0 {
    // 这里会启动一个后台 goroutine，定期清理快照文件(默认 purgeFileInterval 的值为 30s)
    serrc = fileutil.PurgeFile(s.Cfg.SnapDir(), "snap",
       s.Cfg.MaxSnapFiles, purgeFileInterval, s.done)
  }
  if s.Cfg.MaxWALFiles > 0 {
    // 启动一个后台 goroutine，定期清理 WAL 日志文件(默认 purgeFileInterval 的值为 30s)
    werrc = fileutil.PurgeFile(s.Cfg.WALDir(), "wal",
       s.Cfg.MaxWALFiles, purgeFileInterval, s.done)
  }
  // 省略相关的错误处理
}
```

在 fileutil.PurgeFile()函数中会定期查询指定的目录，并统计指定后缀的文件个数，如果文件个数超过指定的上限，则进行清理操作，具体如下所示。

```
func purgeFile(dirname string, suffix string, max uint, interval time.Duration,
     stop <-chan struct{}, purgec chan<- string) <-chan error {
  go func() { // 启动一个后台 goroutine
    for {
      // 查找目标目录下的子文件名称，其中还会对根据文件名称进行排序
      fnames, err := ReadDir(dirname)
      newfnames := make([]string, 0)
```

```
            for _, fname := range fnames { // 根据指定的后缀对文件名称进行过滤
                if strings.HasSuffix(fname, suffix) {
                    newfnames = append(newfnames, fname)
                }
            }
            sort.Strings(newfnames) // 排序
            fnames = newfnames
            for len(newfnames) > int(max) { // 指定后缀的文件个数是否超过上限
                f := filepath.Join(dirname, newfnames[0])
                l, err := TryLockFile(f, os.O_WRONLY, PrivateFileMode) // 获取文件锁
                if err = os.Remove(f); ... // 删除文件，错误处理(略)
                if err = l.Close(); ... // 关闭对应的文件句柄，错误处理(略)
                newfnames = newfnames[1:] // 从 newfnames 中清除指定的文件
            }
            select { // 阻塞指定的时间间隔
            case <-time.After(interval):
            case <-stop:  // 当前节点正在关闭，则退出该 goroutine
                return
            }
        }
    }()
    return errC
}
```

4. linearizableReadLoop

在 EtcdServer.Start()方法中还会启动一个后台 goroutine 负责处理 linearizable Read 相关的请求，相应的逻辑位于 EtcdServer.linearizableReadLoop()方法中。为后续描述方便，我们将该 goroutine 称为“linearizableReadLoop goroutine”。

在开始介绍 linearizableReadLoop()方法之前，先来介绍另一个与 linearizable Read 密切相关的方法，即 EtcdServer.linearizableReadNotify()方法，从图 7-2 中可以看出，当客户端发起 linearizable Read 请求时会先调用 linearizableReadNotify()方法，该方法会通知 linearizableReadLoop goroutine 进行 linearizable Read 相关的处理并阻塞等待其处理结束。

```
v3_server.go  m Range    93    err := s.linearizableReadNotify(ctx)
v3_server.go  m Txn     129    err := s.linearizableReadNotify(ctx)
```

图 7-2

EtcdServer.linearizableReadNotify()方法的具体实现如下所示。

```
func (s *EtcdServer) linearizableReadNotify(ctx context.Context) error {
  nc := s.readNotifier  // 获取 EtcdServer.readNotifier 实例，省略加锁和解锁的代码
  select {
  // 在前面提到的 linearizableReadLoop goroutine 中会阻塞监听 readwaitc 通道，在这里会向
  // readwaitc 通道中写入一个空结构体作为信号，通知 linearizableReadLoop goroutine 开始工作
  case s.readwaitc <- struct{}{}:
  default:
  }

  select {
  // 当 linearizableReadLoop goroutine 发现已应用 Entry 的索引值超过了请求时的 committed
  // Index，则关闭 notifier.c 通道，通知执行读取操作的 goroutine 继续后续读取操作
  case <-nc.c:
    return nc.err
  // 省略其他出现错误场景
  }
}
```

上面使用到的 EtcdServer.readNotifier 字段的类型是 notifier，该结构体中最重要的字段是 c（chan struct{}类型），在 notifier 提供了一个 notify()方法，其中会关闭 notifier.c 通道，在即将介绍的 linearizableReadLoop goroutine 中可以看到该方法的调用。

紧接着来看一下 EtcdServer.linearizableReadLoop()方法，该方法首先会阻塞等待 readwaitc 通道上的信号，readwaitc 通道的功能不再赘述。然后记录当前的 EtcdServer.readNotifier 字段并进行更新，在上面的 linearizableReadNotify()方法中就会监听该 Notifier 实例中封装的通道（c 字段）。之后发送 MsgReadIndex 消息并交由 etcd-raft 模块进行处理，其处理过程在前面的章节已经介绍过了，这里不再展开分析。在前面介绍 raftNode 对 Ready 实例的处理时提到，raftNode 会将其中记录的 ReadState 写入 readStateC 通道中。linearizableReadLoop goroutine 最后会监听 readStateC 通道，等待之前的 MsgReadIndex 消息处理结束。linearizableReadLoop()方法的具体实现如下：

```
func (s *EtcdServer) linearizableReadLoop() {
  var rs raft.ReadState
  for {
    ctx := make([]byte, 8)
    binary.BigEndian.PutUint64(ctx, s.reqIDGen.Next()) // 创建唯一 id
```

```
select {
// 在Client发起一次Linearizable Read时,会向readwaitc通道中写入一个空结构体作为信号,
// 在前面分析 linearizableReadNotify()方法时已经详细说明了 readwaitc 通道的功能
case <-s.readwaitc:
// 监听到 stopping 通道关闭, 则表示当前 EtcdServer 实例正在关闭, 会结束该 goroutine(略)
}
// 记录当前 notifier 实例, 新建 notifier 实例, 并更新 EtcdServer.readNotifier 字段
nextnr := newNotifier()
nr := s.readNotifier
s.readNotifier = nextnr

cctx, cancel := context.WithTimeout(context.Background(), s.Cfg.ReqTimeout())
// 创建并处理 MsgReadIndex 请求, 错误处理(略)
if err := s.r.ReadIndex(cctx, ctx); ...
cancel()
var (
   timeout bool
   done    bool
)
for !timeout && !done { // 检测 MsgReadIndex 请求是否超时, 是否已被处理完
   select {
   // 在 raftNode 处理 Ready 实例时, 会将 Ready.ReadStates 中最后一项写入该通道中
   case rs = <-s.r.readStateC:
      done = bytes.Equal(rs.RequestCtx, ctx) // 在 ReadState.RequestCtx 中携带了请求 id
      if !done {
         plog.Warningf("...") // 忽略乱序的响应, 则输出日志并继续当前 for 循环等待请求对
                              // 应的响应
      }
   case <-time.After(s.Cfg.ReqTimeout()):
      nr.notify(ErrTimeout)  // 在 notifier.err 字段中设置错误信息, 同时关闭
                             // notifier.c 通道
      timeout = true
   // 省略对 stopping 通道的监听
   }
}
if !done {
   continue
}
```

```
        if ai := s.getAppliedIndex(); ai < rs.Index { // ReadState.Index 记录的是 commit
                                                      // Index
            select {
            // 如果当前节点已应用的 Entry 记录未到指定的 committed Index，则需要阻塞等待。这里使用
            // 的 WaitTime.Wait()方法的功能在前面已经介绍过了，不再赘述。后面 EtcdServer.applyAll()
            // 方法处理完待应用 Entry 记录之后，会将已应用 Entry 记录在 WaitTime 中的通道关闭，
            // 则 linearizableReadLoop goroutine 可以得到此信号，继续执行
            case <-s.applyWait.Wait(rs.Index):
            // 省略对 stopping 通道的监听
            }
        }
        nr.notify(nil) // 关闭 notifier.c 通道
    }
}
```

5. apply*

介绍完 EtcdServer.Start()方法中启动的后台 goroutine 的具体功能之后，我们回头来看一下 EtcdServer.start()方法中启动的 run goroutine 的功能。run goroutine 会监听 raftNode.applyc 通道，并调用 EtcdServer.applyAll()方法处理从中读取到的 apply 实例。在前面介绍 raftNode 时提到，在 apply 实例中封装了待应用的 Entry 记录、待应用的快照数据和 notifyc 通道。

➢ applySnapshot

在 EtcdServer.applyAll()方法中，首先会调用 EtcdServer.applySnapshot()方法处理 apply 实例中的快照数据。EtcdServer.applySnapshot()方法会先等待 raftNode 将快照数据持久化到磁盘中，之后根据快照元数据查找 BoltDB 数据库文件并重建 Backend 实例，最后根据重建后的存储更新本地 RaftCluster 实例。applySnapshot()方法的具体实现如下所示。

```
func (s *EtcdServer) applySnapshot(ep *etcdProgress, apply *apply) {
    // 检测待应用的快照数据是否为空，如果为空则直接返回(略)
    if apply.snapshot.Metadata.Index <= ep.appliedi {
        // 如果该快照中最后一条 Entry 的索引值小于当前节点已应用 Entry 索引值，则程序异常结束(略)
    }
    // 读者可以回顾前面对 raftNode 的分析，raftNode 在将快照数据写入磁盘文件之后，会向 notifyc
    // 通道中写入一个空结构体作为信号，这里会阻塞等待该信号
    <-apply.notifyc
    // 根据快照信息查找对应的 BoltDB 数据库文件，并创建新的 Backend 实例，错误处理(略)
```

```
newbe, err := openSnapshotBackend(s.Cfg, s.snapshotter, apply.snapshot)

// 因为在 store.restore()方法中除了恢复内存索引，还会重新绑定键值对与对应的 Lease，
// 所以需要先恢复 EtcdServer.lessor，再恢复 EtcdServer.kv 字段
if s.lessor != nil {
   s.lessor.Recover(newbe, func() lease.TxnDelete { return s.kv.Write() })
}
if err := s.kv.Restore(newbe); ...
// 重置 EtcdServer.consistIndex 字段
s.consistIndex.setConsistentIndex(s.kv.ConsistentIndex()).
oldbe := s.be
go func() {
   // 因为此时可能还有事务在执行，关闭旧 Backend 实例可能会被阻塞，所以这里启动一个后
   // 台 goroutine 用来关闭 Backend 实例
   if err := oldbe.Close(); ...
}()
s.be = newbe // 更新 EtcdServer 实例中使用的 Backend 实例
// 恢复 EtcdServer 中的 alarmStore 和 authStore，它们分别对应 BoltDB 中的 alarm Bucket
// 和 auth Bukcet，后面会介绍 alarmStore 和 authStore 的作用
if err := s.restoreAlarms(); ...
if s.authStore != nil {
   s.authStore.Recover(newbe)
}

// 恢复 v2 版本存储，该恢复过程在前面已经介绍过了，这里不再赘述
if err := s.store.Recovery(apply.snapshot.Data); ...
s.cluster.SetBackend(s.be) // 设置 RaftCluster.backend 实例
s.cluster.Recover(api.UpdateCapability) // 恢复本地的集群信息，不再展开介绍
// 清空 Transport 中所有 Peer 实例，并根据恢复后的 RaftCluster 实例重新添加
s.r.transport.RemoveAllPeers()
for _, m := range s.cluster.Members() {
   if m.ID == s.ID() {
      continue
   }
   s.r.transport.AddPeer(m.ID, m.PeerURLs)
}

// 更新 etcdProgress，其中涉及已应用 Entry 记录的 Term 值、Index 值和快照相关信息
```

```
    ep.appliedt = apply.snapshot.Metadata.Term
    ep.appliedi = apply.snapshot.Metadata.Index
    ep.snapi = ep.appliedi
    ep.confState = apply.snapshot.Metadata.ConfState
}
```

➢ applyEntries

应用完快照数据之后，run goroutine 紧接着会调用 EtcdServer.applyEntries()方法处理待应用的 Entry 记录，具体实现如下所示。

```
func (s *EtcdServer) applyEntries(ep *etcdProgress, apply *apply) {
    // 检测是否存在待应用的 Entry 记录，如果为空则直接返回(略)
    firsti := apply.entries[0].Index
    if firsti > ep.appliedi+1 {// 检测待应用的第一条 Entry 记录是否合法(略) }
    var ents []raftpb.Entry
    if ep.appliedi+1-firsti < uint64(len(apply.entries)) {
        // 忽略已应用的 Entry 记录，只保留未应用的 Entry 记录
        ents = apply.entries[ep.appliedi+1-firsti:]
    }
    if len(ents) == 0 {
        return
    }
    var shouldstop bool
    // 调用 apply()方法应用 ents 中的 Entry 记录
    if ep.appliedt, ep.appliedi, shouldstop = s.apply(ents, &ep.confState); shouldstop {
        go s.stopWithDelay(10*100*time.Millisecond, fmt.Errorf("..."))
    }
}
```

上述 applyEntries()方法最终会通过 EtcdServer.apply()方法完成 Entry 记录的应用，在 apply()方法中会遍历 ents 中的全部 Entry 记录，并根据 Entry 的类型进行不同的处理。apply()方法的具体实现如下：

```
func (s *EtcdServer) apply(es []raftpb.Entry, confState *raftpb.ConfState)
        (appliedt uint64, appliedi uint64, shouldStop bool) {
    for i := range es { // 遍历待应用的 Entry 记录
        e := es[i]
        switch e.Type { // 根据 Entry 记录的不同类型，进行不同的处理
```

```
        case raftpb.EntryNormal:
            s.applyEntryNormal(&e)
        case raftpb.EntryConfChange:
            // 更新 EtcdServer.consistIndex，其中保存了当前节点应用的最后一条 Entry 记录的索引值
            if e.Index > s.consistIndex.ConsistentIndex() {
                s.consistIndex.setConsistentIndex(e.Index)
            }
            var cc raftpb.ConfChange
            pbutil.MustUnmarshal(&cc, e.Data) // 将 Entry.Data 反序列化成 ConfChange 实例
            // 调用 applyConfChange()方法处理 ConfChange，注意返回值 removedSelf，当它为 true
            // 时表示将当前节点从集群中移除
            removedSelf, err := s.applyConfChange(cc, confState)
            s.setAppliedIndex(e.Index) // 更新 EtcdServer.appliedIndex 字段
            shouldStop = shouldStop || removedSelf
            // 在后面介绍 AddMember()、UpdateMember()、RemoveMember()等方法时会看到，它们都会
            // 阻塞监听 ConfChange 对应的通道，Wait 的原理不再赘述。这里在对应的 Entry 处理完成之后，
            // 会关闭对应的通道，通知监听的 goroutine
            s.w.Trigger(cc.ID, &confChangeResponse{s.cluster.Members(), err})
        default:
            plog.Panicf("entry type should be either EntryNormal or EntryConfChange")
        }
        atomic.StoreUint64(&s.r.index, e.Index) // 更新 raftNode 的 index 字段和 term 字段返回值
        atomic.StoreUint64(&s.r.term, e.Term)
        appliedt = e.Term  // 更新 appliedt、appliedi 返回值
        appliedi = e.Index
    }
    return appliedt, appliedi, shouldStop // shouldStop 标识当前节点是否已经从集群中移除
}
```

applyEntryNormal

这里我们介绍一下 EtcdServer.applyEntryNormal()方法处理 EntryNormal 记录的具体过程。applyEntryNormal()方法首先会尝试将 Entry.Data 反序列化成 InternalRaftRequest 实例，如果失败，则将其反序列化成 etcdserverpb.Request 实例，之后根据反序列化的结果调用 EtcdServer 的相应方法进行处理（具体的解析过程后面中会展开详细介绍），最后将处理结果写入 Entry 对应的通道中。applyEntryNormal()方法的具体实现如下：

```
func (s *EtcdServer) applyEntryNormal(e *raftpb.Entry) {
```

```
shouldApplyV3 := false
if e.Index > s.consistIndex.ConsistentIndex() {
   s.consistIndex.setConsistentIndex(e.Index) // 更新 EtcdServer.consistIndex 记录
                                              // 的索引值
   shouldApplyV3 = true
}
defer s.setAppliedIndex(e.Index) // 方法结束时更新 EtcdServer.appliedIndex 字段记录的
                                 // 索引值

if len(e.Data) == 0 { // 空的 Entry 记录只会在 Leader 选举结束时出现
   if s.isLeader() { // 如果当前节点为 Leader，则晋升其 lessor 实例
      s.lessor.Promote(s.Cfg.electionTimeout())
   }
   return
}

// 忽略没有数据的 Entry 记录(略)
var raftReq pb.InternalRaftRequest
// 尝试将 Entry.Data 反序列化成 InternalRaftRequest 实例，InternalRaftRequest 中封装了所
// 有类型的 Client 请求
if !pbutil.MaybeUnmarshal(&raftReq, e.Data) {
   var r pb.Request
   // 兼容性处理，如果上述序列化失败，则将 Entry.Date 反序列化成 pb.Request 实例
   pbutil.MustUnmarshal(&r, e.Data)
   // 调用 EtcdServer.applyV2Request()方法进行处理，在 applyV2Request()方法中，
   // 会根据请求的类型，调用不同的方法进行处理。处理结束后会将结果写入 Wait 中记录对应的通道中，
   // 然后关闭该通道，Wait 的原理前面已经介绍过了，这里不再赘述。
   s.w.Trigger(r.ID, s.applyV2Request(&r))
   return
}
if raftReq.V2 != nil { // 上述序列化成功，且是 v2 版本的请求，调用 applyV2Request()方法处理
   req := raftReq.V2
   s.w.Trigger(req.ID, s.applyV2Request(req)) // 关闭 Wait 中记录的该 Entry 对应的通道
   return
}
// 下面是对 v3 版本请求的处理
id := raftReq.ID // 获取请求 id
var ar *applyResult
```

```
    needResult := s.w.IsRegistered(id) // 检测该请求是否需要进行响应
    if needResult || !noSideEffect(&raftReq) {
        if !needResult && raftReq.Txn != nil {
            removeNeedlessRangeReqs(raftReq.Txn)
        }
        // 调用 applyV3.Apply()方法处理该 Entry，其中会根据请求的类型选择不同的方法进行处理
        ar = s.applyV3.Apply(&raftReq)
    }
    // 返回结果 ar(applyResult 类型)为 nil，直接返回(略)
    // 如果返回了 ErrNoSpace 错误，则表示底层的 Backend 已经没有足够的空间，如果是第一次出现这种情
    // 况，则在后面立即启动一个后台 goroutine，并调用 EtcdServer.raftRequest()方法发送
    // AlarmRequest 请求，当前其他节点收到该请求时，会停止后续的 PUT 操作，后面介绍 applyV3 接口
    // 及其实现时详细分析
    if ar.err != ErrNoSpace || len(s.alarmStore.Get(pb.AlarmType_NOSPACE)) > 0 {
        s.w.Trigger(id, ar)  // 将上述处理结果写入对应的通道中，然后将对应通道关闭
        return
    }
    s.goAttach(func() {  // 第一次出现 ErrNoSpace 错误
        a := &pb.AlarmRequest{  // 创建 AlarmRequest
            MemberID: uint64(s.ID()),
            Action:   pb.AlarmRequest_ACTIVATE,
            Alarm:    pb.AlarmType_NOSPACE,
        }
        // 将 AlarmRequest 请求封装成 MsgProp 消息，并发送到集群中，后面会详细介绍该方法的实现
        s.raftRequest(s.ctx, pb.InternalRaftRequest{Alarm: a})
        s.w.Trigger(id, ar) // 将上述处理结果写入对应的通道中，然后将对应通道关闭
    })
}
```

applyConfChange

EntryConfChange 类型的 Entry 记录主要是通过 EtcdServer.applyConfChange()方法进行处理的，在 applyConfChange()方法中会根据 ConfChange 的类型进行分类处理，其具体实现如下：

```
func (s *EtcdServer) applyConfChange(cc raftpb.ConfChange, confState *raftpb.ConfState)
        (bool, error) {
    // 在开始进行节点修改之前，先调用 ValidateConfigurationChange()方法进行检测，
    // 后面会展开介绍该方法的具体实现，错误处理(略)
    if err := s.cluster.ValidateConfigurationChange(cc); ...
```

```
// 将 ConfChange 交给 etcd-raft 模块进行处理，其中会根据 ConfChange 的类型进行分类处理，
// etcd-raft 模块对 ConfChange 的处理在前已经详细介绍路过了，这里不再展开介绍
// 这里的返回值的 ConfState 中记录了集群中最新的节点 id
*confState = *s.r.ApplyConfChange(cc)
switch cc.Type { // 根据请求的类型进行相应的处理
case raftpb.ConfChangeAddNode:
   // 将 ConfChange.Context 中的数据反序列化成 Member 实例，错误处理(略)
   m := new(membership.Member)
   if err := json.Unmarshal(cc.Context, m); ...
   s.cluster.AddMember(m) // 将新 Member 实例添加到本地 RaftCluster 中
   if m.ID != s.id {
      // 如果添加的是远端节点，则需要在 Transport 中添加对应的 Peer 实例，Transport 的实现前
      // 面已经详细介绍过了，这里不再展开介绍
      s.r.transport.AddPeer(m.ID, m.PeerURLs)
   }
case raftpb.ConfChangeRemoveNode:
   id := types.ID(cc.NodeID)
   s.cluster.RemoveMember(id) // 从 RaftCluster 中删除指定的 Member 实例
   if id == s.id {
      return true, nil // 如果移除的是当前节点则返回 true
   }
   s.r.transport.RemovePeer(id) // 若移除远端节点，则需要从 Transporter 中删除对应的 Peer 实例
case raftpb.ConfChangeUpdateNode:
   // 将 ConfChange.Context 中的数据反序列化成 Member 实例，错误处理(略)
   m := new(membership.Member)
   if err := json.Unmarshal(cc.Context, m); err != nil ...
   // 更新本地 RaftCluster 实例中相应的 Member 实例
   s.cluster.UpdateRaftAttributes(m.ID, m.RaftAttributes)
   if m.ID != s.id { // 如果是更新的是远端节点，则需要更新 Transporter 中对应的 Peer 实例
      s.r.transport.UpdatePeer(m.ID, m.PeerURLs)
   }
}
return false, nil
}
```

在上面的 RaftCluster.ValidateConfigurationChange()方法中，会检测待修改的节点信息是否合法，例如，新增节点提供的 URL 是否与集群中现有节点的 URL 冲突，待删除的节点是否存在，等等。下面简单分析 RaftCluster.ValidateConfigurationChange()方法的实现：

仅供非商业用途或交流学习使用


```
func (c *RaftCluster) ValidateConfigurationChange(cc raftpb.ConfChange) error {
  // 从 v2 存储中获取当前集群中存在的节点信息(members 变量)，以及已移除的节点信息(removed 变量)
  // 这里的 members 的类型是 map[types.ID]*Member，removed 的类型是 map[types.ID]bool
  // members 的内容实际上记录在 v2 存储中的“/0/members”节点下，removed 的内容记录在 v2 存
  // 储的“/0/removed_members”节点下，membersFromStore()方法就是从这两个节点下查找相应信
  // 息的，其具体实现不再展开详述，感兴趣的读者可以参考代码
  members, removed := membersFromStore(c.store)
  id := types.ID(cc.NodeID) // 从 ConfChange 实例中获取待操作的节点 id
  // 检测该节点是否在 removed 中，如果存在，则返回错误(略)
  switch cc.Type { // 下面根据 ConfChange 的类型进行分类处理
  case raftpb.ConfChangeAddNode:
    // 检测新增节点是否在 members 中，如果存在，则返回错误(略)
    // 遍历当前集群中全部的 Member 实例，并将每个节点暴露的 URL 记录到 map 中
    urls := make(map[string]bool)
    for _, m := range members {
      for _, u := range m.PeerURLs {
        urls[u] = true
      }
    }
    m := new(Member) // 将 ConfChange.Context 反序列化成 Member 实例，错误处理(略)
    if err := json.Unmarshal(cc.Context, m); err != nil ...
    // 遍历新增节点提供的 URL 是否与集群中已有节点提供的 URL 地址冲突
    for _, u := range m.PeerURLs {
      if urls[u] {
        return ErrPeerURLexists
      }
    }
  case raftpb.ConfChangeRemoveNode:
    // 检测待删除节点是否存在于 members 中(略)
  case raftpb.ConfChangeUpdateNode:
    // 检测待删除节点是否存在于 members 中（略）
    // 检测节点更新之后提供的 URL 地址是否与集群中其他节点提供的 URL 地址冲突，
    // 具体检测过程与新增节点类似(略)
  default: // 未知 ConfChange 类型，输出错误日志(略)
  }
  return nil
}
```

仅供非商业用途或交流学习使用

到此为止，EtcdServer.applyEntries()方法对待应用的 Entry 的处理流程就介绍完了。下面我们回到 EtcdServer.applyAll()方法继续进行分析。

➢ applyAll

在EtcdServer.applyAll()方法中首先调用applySnapshot()方法处理apply实例中记录的快照数据，然后调用 applyEntries()方法处理 apply 实例中的 Entry 记录，之后根据 apply 实例的处理结果检测是否需要生成新的快照文件，最后处理 MsgSnap 消息。EtcdServer.applyAll()方法的具体实现如下：

```
func (s *EtcdServer) applyAll(ep *etcdProgress, apply *apply) {
    s.applySnapshot(ep, apply) // 调用 applySnapshot()方法处理 apply 实例中记录的快照数据
    s.applyEntries(ep, apply) // 调用 applyEntries()方法处理 apply 实例中的 Entry 记录
    // 在前面的介绍中提到，etcdProgress.appliedi 记录了已应用 Entry 的索引值。这里通过调用
    // WaitTime.Trigger()方法，将 id 小于 etcdProgress.appliedi 的 Entry 对应的通道全部关闭，
    // 这样就可以通知其他监听通道的 goroutine，例如，前面介绍的 linearizableReadLoop goroutine
    s.applyWait.Trigger(ep.appliedi)
    // 读者可以回顾前面对 raftNode 的分析，当 Ready 处理基本完成时，会向 notifyc 通道中写入一个信
    // 号，通知当前 goroutine 去检测是否需要生成快照
    <-apply.notifyc
    s.triggerSnapshot(ep) // 根据当前状态决定是否触发快照的生成，后面展开介绍
    select {
    // 在 raftNode 中处理 Ready 实例时，如果并没有直接发送 MsgSnap 消息，而是将其写入 msgSnapC 通
    // 道中，这里会读取 msgSnapC 通道，并完成快照数据的发送
    case m := <-s.r.msgSnapC:
        // 将 v2 存储的快照数据和 v3 存储的数据合并成完整的快照数据
        merged := s.createMergedSnapshotMessage(m, ep.appliedt, ep.appliedi, ep.confState)
        s.sendMergedSnap(merged)  // 发送快照数据
    default:
    }
}
```

先简单介绍 EtcdServer.triggerSnapshot()方法对是否需要生成新快照文件的判定，其具体实现如下：

```
func (s *EtcdServer) triggerSnapshot(ep *etcdProgress) {
    if ep.appliedi-ep.snapi <= s.snapCount {
        return // 连续应用一定量的 Entry 记录，会触发快照的生成(snapCount 默认为 100000 条)
```

仅供非商业用途或交流学习使用


```
    }
    s.snapshot(ep.appliedi, ep.confState) // 创建新的快照文件
    ep.snapi = ep.appliedi // 更新 etcdProgress.snapi
}
```

EtcdServer.snapshot()方法是真正生成快照文件的地方，其中会启动一个单独的后台 goroutine 来完成新快照文件的生成，主要是序列化 v2 存储中的数据并持久化到文件中，触发相应的压缩操作，其具体实现如下：

```
func (s *EtcdServer) snapshot(snapi uint64, confState raftpb.ConfState) {
    clone := s.store.Clone() // 复制 v2 存储
    s.KV().Commit() // 提交 v3 存储中当前等待读写事务
    s.goAttach(func() {
        d, err := clone.SaveNoCopy() // 将 v2 存储序列化成 JSON 数据，错误处理(略)
        // 将上述快照数据和元数据更新到 etcd-raft 模块中的 MemoryStorage 中，
        // 并且返回 Snapshot 实例(即 MemoryStorage.snapshot 字段)，错误处理(略)
        snap, err := s.r.raftStorage.CreateSnapshot(snapi, &confState, d)
        // 将 v2 存储的快照数据记录到磁盘中，该过程涉及在 WAL 日志文件中记录快照元数据及写入 snap 文
        // 件等操作，具体实现在前面已经介绍过了，这里不再展开。错误处理(略)
        if err = s.r.storage.SaveSnap(snap); err != nil ...
        // 如果当前还存在已发送但未响应的快照消息，则不能进行后续的压缩操作，如果进行了后续的压缩，
        // 则可能导致 Follower 节点再次无法追赶上 Leader 节点，从而需要再次发送快照数据
        if atomic.LoadInt64(&s.inflightSnapshots) != 0 {
            return
        }
        // 为了防止集群中存在比较慢的 Follower 节点，保留 5000 条 Entry 记录不压缩
        compacti := uint64(1)
        if snapi > numberOfCatchUpEntries {
            compacti = snapi - numberOfCatchUpEntries
        }
        // 压缩 MemoryStorage 中的指定位置之前的全部 Entry 记录，不再展开
        err = s.r.raftStorage.Compact(compacti)
    })
}
```

介绍完 snap 快照文件的生成，再来分析对 msgSnapC 通道的处理。首先分析 EtcdServer.createMergedSnapshotMessage()方法，该方法中会将 v2 版本存储和 v3 版本存储封装成 snap.Message 实例，具体实现如下所示。

仅供非商业用途或交流学习使用


```
func (s *EtcdServer) createMergedSnapshotMessage(m raftpb.Message, snapt,
    snapi uint64, confState raftpb.ConfState) snap.Message {
  clone := s.store.Clone() // 复制一份 v2 存储的数据，并转换成 JSON 格式，错误处理(略)
  d, err := clone.SaveNoCopy()
  s.KV().Commit() // 提交 v3 存储中当前的读写事务
  // 获取 v3 存储快照，其实就是对 BoltDB 数据库进行快照，前面介绍过该方法的具体实现，这里不再展开描述
  dbsnap := s.be.Snapshot()
  rc := newSnapshotReaderCloser(dbsnap).
  // 将 v2 存储的快照数据和相关元数据写入 raftpb.Snapshot 实例中
  snapshot := raftpb.Snapshot{
    Metadata: raftpb.SnapshotMetadata{
      Index:     snapi,
      Term:      snapt,
      ConfState: confState,
    },
    Data: d,
  }
  m.Snapshot = snapshot  // 将 raftpb.Snapshot 实例封装到 MsgSnap 消息中
  // 将消息 MsgSnap 消息和 v3 存储中的数据封装成 snap.Message 实例返回
  return *snap.NewMessage(m, rc, dbsnap.Size())
}
```

创建完 snap.Message 实例之后会调用 EtcdServer.sendMergedSnap()方法将其发送到指定节点，其具体实现如下：

```
func (s *EtcdServer) sendMergedSnap(merged snap.Message) {
  // 递增 inflightSnapshots 字段，它表示已发送但未收到响应的快照消息个数
  atomic.AddInt64(&s.inflightSnapshots, 1)
  // 发送 snap.Message 消息，底层会启动单独的后台 goroutine，通过 snapshotSender 完成发送，
  // 具体的实现在前面已经详细介绍过了，这里不再赘述
  s.r.transport.SendSnapshot(merged)
  s.goAttach(func() { // 启动一个后台 goroutine 监听该快照消息是否发送完成
    select {
    case ok := <-merged.CloseNotify():
      if ok {
        select {
        case <-time.After(releaseDelayAfterSnapshot): // 默认阻塞等待 30 秒
        }
```

仅供非商业用途或交流学习使用


```
            }
            // snap.Message 消息发送完成(或是超时)之后会递减 inflightSnapshots 值,
            // 当 inflightSnapshots 递减到 0 时, 前面对 MemoryStorage 的压缩才能执行
            atomic.AddInt64(&s.inflightSnapshots, -1)
        }
    })
}
```

6. SYNC 消息

在 EtcdServer 启动的 run goroutine 中，另一个需要介绍的功能是定时发送 SYNC 消息。SYNC 消息的主要作用是定义通知节点，清理过期 v2 存储中的过期节点。在前面介绍 raftReadyHandler 时提到，当节点成为 Leader 时，会调用 setSyncC()回调函数设置一个定时器，用来触发 SYNC 消息的定期发送。

而在 run goroutine 中会通过 getSyncC()函数监听该定时器，当定时器到期时会调用 EtcdServer.sync()方法发送 SYNC 消息，该方法的具体如下：

```
func (s *EtcdServer) sync(timeout time.Duration) {
    req := pb.Request{ // 创建 SYNC 消息
        Method: "SYNC",
        ID:     s.reqIDGen.Next(),
        Time:   time.Now().UnixNano(),
    }
    data := pbutil.MustMarshal(&req) // 序列化 SYNC 消息
    ctx, cancel := context.WithTimeout(s.ctx, timeout)
    s.goAttach(func() { // 启动一个后台 goroutine, 发送 SYNC(MsgProp)消息
        s.r.Propose(ctx, data)
        cancel()
    })
}
```

7. expiredLeaseC

前面的章节中提到，在 lessor 实例初始化的过程中会启动一个后台 goroutine，该后台 goroutine 会定时扫描 lessor.leaseMap 字段，并将过期的 Lease 实例写入 expireLeaseC 通道中等待处理。

在 EtcdServer 启动的 run goroutine 中，最后一个需要介绍的功能就是监听 expireLeaseC 通道，当监听到过期 Lease 实例之后会启动一个后台 goroutine 进行处理，相关代码片段的实现如下：

仅供非商业用途或交流学习使用


```
case leases := <-expiredLeaseC: // 监听 expiredLeaseC 通道
  s.goAttach(func() { // 启动单独的 goroutine
    c := make(chan struct{}, maxPendingRevokes) // 用于限流，上限是 16
    for _, lease := range leases { // 遍历过期的 Lease 实例
      select {
      case c <- struct{}{}: // 向 c 通道中添加一个空结构体
      case <-s.stopping:
        return
      }
      lid := lease.ID
      s.goAttach(func() { // 启动一个后台线程，完成指定 Lease 的撤销
        s.LeaseRevoke(s.ctx, &pb.LeaseRevokeRequest{ID: int64(lid)})
        leaseExpired.Inc()
        <-c
      })
    }
  })
```

接下来介绍 EtcdServer.LeaseRevoke()方法，其中会将 LeaseRevokeRequest 请求中的信息封装成 MsgProp 消息，并发送到集群中的其他节点，具体实现如下所示。

```
func (s *EtcdServer) LeaseRevoke(ctx context.Context, r *pb.LeaseRevokeRequest)
    (*pb.LeaseRevokeResponse, error) {
  // 将 LeaseRevokeRequest 封装成 InternalRaftRequest，并调用 raftRequestOnce()方法，错
  // 误处理(略)
  resp, err := s.raftRequestOnce(ctx, pb.InternalRaftRequest{LeaseRevoke: r})
  return resp.(*pb.LeaseRevokeResponse), nil
}
```

其中的 raftRequestOnce()方法比较简单（前面发送 AlarmRequest 请求时使用的 raftRequest()方法，底层就是调用该方法实现的），其中直接调用了 processInternalRaftRequestOnce()方法，下面直接分析 processInternalRaftRequestOnce()方法，如下所示。

```
func (s *EtcdServer) processInternalRaftRequestOnce(ctx context.Context,
    r pb.InternalRaftRequest) (*applyResult, error) {
  // 检测当前是否有大量已提交但未应用的 Entry 记录，如果有，则返回错误实现限流(略)
  r.Header = &pb.RequestHeader{ // 为请求生成 id
    ID: s.reqIDGen.Next(),
```

仅供非商业用途或交流学习使用


```
    }
    // 获取权限信息，记录到请求头中，错误处理(略)
    authInfo, err := s.AuthInfoFromCtx(ctx)
    if authInfo != nil {
       r.Header.Username = authInfo.Username
       r.Header.AuthRevision = authInfo.Revision
    }
    data, err := r.Marshal() // 将 InternalRaftRequest 请求序列化，错误处理(略)
    // 检测序列化后的长度(略)
    id := r.ID
    if id == 0 {
       id = r.Header.ID
    }
    ch := s.w.Register(id) // 在 EtcdServer.wait 中为该请求注册相应的 ch 通道
    // 将 InternalRaftRequest 序列化后的数据封装为 Entry，之后生成 MsgProp 消息并发送出去，具体
    // 的发送过程在前介绍过了，这里不再展开描述
    s.r.Propose(cctx, data)
    select {
    case x := <-ch: // 阻塞等待上面发送的 Entry 被应用
       return x.(*applyResult), nil
    // 省略其他场景的处理
    }
}
```

待上述 LeaseRevokeRequest 请求对应的 Entry 记录成功经过 Raft 协议处理之后，会经过前面介绍的 applyEntryNormal()方法的处理，其中会调用 applierV3.LeaseRevoke()方法，该方法实际上是调用 EtcdServer.lessor.LeaseRevoke()方法，该方法会删除对应的 Lease 实例及其绑定的键值对，其具体实现在前面的章节中介绍过了，这里不再赘述。

7.4 ApplierV2 和applierV3

7.4.1 ApplierV2

通过上一节的分析我们知道，在 EtcdServer.applyEntryNormal()方法中完成了对 EntryNormal 类型记录的处理，其中会调用 applyV2Request()方法处理 v2 版本的消息。在 applyV2Request()方法中会根据请求的类型进行分类处理，该方法具体实现如下所示。

仅供非商业用途或交流学习使用


```
func (s *EtcdServer) applyV2Request(r *pb.Request) Response {
  toTTLOptions(r)
  switch r.Method {
  case "POST":
    return s.applyV2.Post(r)
  case "PUT":
    return s.applyV2.Put(r)
  case "DELETE":
    return s.applyV2.Delete(r)
  case "QGET":
    return s.applyV2.QGet(r)
  case "SYNC":
    return s.applyV2.Sync(r)
  default: // 未知 Method，返回错误信息(略)
  }
}
```

在 ApplierV2 接口中定义了处理 v2 版本消息的方法，即 applyV2Request()方法中调用的 Post()、Put()、Delete()、QGet()、Sync()等方法。结构体 applierV2store 实现 ApplierV2 接口，EtcdServer.applyV2 字段就是 applierV2store 类型。applierV2store 中最重要的就是 store 字段，它指向当前节点的 v2 存储。

首先看一下 applierV2store.Put()方法，其中根据请求内容进行分类处理，具体实现如下所示。

```
func (a *applierV2store) Put(r *pb.Request) Response {
  // 根据请求内容创建 TTLOptionSet 实例，其中会设置节点的超时时间、此次请求是否为刷新操作
  ttlOptions := toTTLOptions(r)
  exists, existsSet := pbutil.GetBool(r.PrevExist) // 修改节点之前是否存在
  switch {
  case existsSet:
    if exists {
      if r.PrevIndex == 0 && r.PrevValue == "" {
        // 未提供 PreIndex 和 PreValue 信息，则直接调用 Update()方法更新节点值
        return toResponse(a.store.Update(r.Path, r.Val, ttlOptions))
      }
      // 提供了 PreIndex 和 PreValue 信息，则调用 CompareAndSwap()方法更新节点值
      // CompareAndSwap()方法会比较当前节点的 PreIndex 和 PreValue 是否与此次操作提供提供
      // 的值相同，然后决定是否修改
```

仅供非商业用途或交流学习使用


```
            return toResponse(a.store.CompareAndSwap(r.Path, r.PrevValue, r.PrevIndex,
                        r.Val, ttlOptions))
        }
        return toResponse(a.store.Create(r.Path, r.Dir, r.Val, false, ttlOptions))
    case r.PrevIndex > 0 || r.PrevValue != "":
        return toResponse(a.store.CompareAndSwap(r.Path, r.PrevValue, r.PrevIndex,
                r.Val, ttlOptions))
    default:
        // 操作的节点是“/0/members”下的节点集群成员信息节点，注意，不会修改 v2 存储中对应的节点
        if storeMemberAttributeRegexp.MatchString(r.Path) {
            // 从节点的路径信息中解析得到节点 id
            id := membership.MustParseMemberIDFromKey(path.Dir(r.Path))
            var attr membership.Attributes // 将节点值反序列化
            if a.cluster != nil { // 更新 RaftCluster 中对应节点的信息
                a.cluster.UpdateAttributes(id, attr)
            }
            return Response{}
        }
        if r.Path == membership.StoreClusterVersionKey() { // 操作的节点是“/0/version”
            if a.cluster != nil { // 更新 RaftCluster 中的版本信息
                a.cluster.SetVersion(semver.Must(semver.NewVersion(r.Val)),
                    api.UpdateCapability)
            }
            return Response{}
        }
        // 如果不是上述两种节点，则直接调用 Set()方法更新对应节点值
        return toResponse(a.store.Set(r.Path, r.Dir, r.Val, ttlOptions))
    }
}
```

applierV2store 中的 Post()、Delete()、QGet()方法实现比较简单，分别调用了 store 的 Create()、Delete()（或是 CompareAndDelete()）、Get()方法，这里不再展开介绍。在前面介绍 EtcdServer 启动的 run goroutine 时提到，它会定期向集群中发送 SYNC 消息，当消息经过 etcd-raft 模块处理并最终应用时，就是由 applierV2store.Sync()方法处理的，其中会调用 store.DeleteExpiredKeys()方法清理 v2 存储中已经过期的节点，该清理过程在前面已经介绍过了，这里不再赘述。

仅供非商业用途或交流学习使用

7.4.2　applierV3

对应于上一节介绍的 ApplierV2 接口，applierV3 接口定义了在 v3 存储之上应用 Entry 记录的功能。相较于 ApplierV2 接口，applierV3 接口提供了更加强大且复杂的功能。applierV3 接口中定义的核心方法如下：

```
type applierV3 interface {
  // 这是在前面介绍 EtcdServer.applyEntryNormal()方法时见到的方法，其中会根据
  // InternalRaftRequest 的类型调用下面不同的方法进行处理
  Apply(r *pb.InternalRaftRequest) *applyResult
  // 分别对应于 v3 存储中的同名方法，不再一一介绍
  Put(txn mvcc.TxnWrite, p *pb.PutRequest) (*pb.PutResponse, error)
  Range(txn mvcc.TxnRead, r *pb.RangeRequest) (*pb.RangeResponse, error)
  DeleteRange(txn mvcc.TxnWrite, dr *pb.DeleteRangeRequest)
       (*pb.DeleteRangeResponse, error)
  Txn(rt *pb.TxnRequest) (*pb.TxnResponse, error)
  Compaction(compaction *pb.CompactionRequest)
       (*pb.CompactionResponse, <-chan struct{}, error)
  // 其他扩展方法比较多，后面遇到时再详细介绍
}
```

这里实现了 applierV3 接口的结构体比较多，如图 7-3 所示，其中 quotaApplierV3、applierV3Capped、authApplierV3 都是在 applierV3backend 的基础上进行的扩展，在结构体 applierV3backend 中实现了真正应用 Entry 记录并完成相应的存储功能。

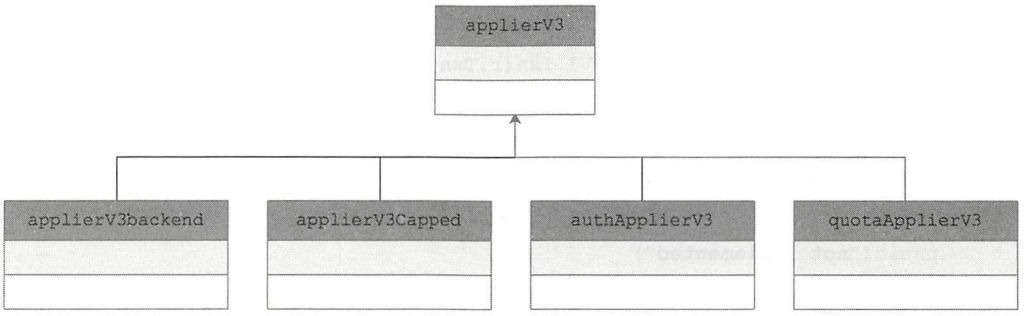


图 7-3

这里简单介绍 applierV3backend 之外的其他 applierV3 接口实现的功能。

- **quotaApplierV3**：在 applierV3backend 的基础上提供了限流功能，即底层的 BoltDB 数

据库文件的大小增大到上限之后，就会触发限流操作。

- **applierV3Capped**：在 quotaApplierV3 触发限流操作之后，就会创建 applierV3Capped 实例替换 EtcdServer 当前使用的 applierV3 接口实现。通过 applierV3Capped 实例执行任何写入操作都会失败（例如，Put()方法等），这样就可以保证底层存储中的数据量不再增加。
- **authApplierV3**：在 applierV3backend 的基础上扩展出了权限控制的功能。

1. applierV3backend

applierV3backend 结构体是 applierV3 接口的核心实现，在 applierV3backend 中只有一个指向当前 EtcdServer 实例的字段。在 applierV3backend.Apply()方法中会根据请求的类型进行分类处理，大致实现如下：

```
func (a *applierV3backend) Apply(r *pb.InternalRaftRequest) *applyResult {
  ar := &applyResult{}
  // 需要注意的是，这里调用的是 EtcdServer.applyV3 对应的方法，而不是直接调用 applierV3backend 实
  // 例的对应方法。在 EtcdServer 中，applyV3Base 字段指向了 applierV3backend 实例，而 applyV3
  // 字段则是在 applyV3Base 基础之上的扩展
  switch {
  case r.Range != nil:
    ar.resp, ar.err = a.s.applyV3.Range(nil, r.Range)
  case r.Put != nil:
    ar.resp, ar.err = a.s.applyV3.Put(nil, r.Put)
  case r.DeleteRange != nil:
    ar.resp, ar.err = a.s.applyV3.DeleteRange(nil, r.DeleteRange)
  case r.Txn != nil:
    ar.resp, ar.err = a.s.applyV3.Txn(r.Txn)
  case r.Compaction != nil:
    ar.resp, ar.physc, ar.err = a.s.applyV3.Compaction(r.Compaction)
  // 省略其他扩展方法的调用，后面遇到时会详细分析
  default:
    panic("not implemented")
  }
  return ar
}
```

本节后面的部分，我们将逐个分析 applierV3backend 中比较核心的方法。

➢ Put

在开始分析 applierV3backend.Put()方法之前，先介绍一下 PutRequest 封装的信息，其中各个字段的含义如下所示。

- Key（[]byte 类型）：此次请求的目标 Key 值。
- Value（[]byte 类型）：此次请求更新后的 Value 值。
- PreKv（bool 类型）：如果该字段设置为 true，则在进行修改之前会获取原始的键值对，并将其封装到 PutResponse 中。
- IgnoreValue（bool 类型）：如果该字段设置为 true，则表示此次请求不会更新 Value 值。如果此次请求中指定的 Key 不存在，则返回 ErrKeyNotFound 错误。
- IgnoreLease（bool 类型）：如果该字段设置为 true，则表示此次请求不会更新对应键值对的 lease。如果此次请求中指定的 Key 不存在，则返回 ErrKeyNotFound 错误。

接下来分析 applierV3backend.Put()方法的具体实现，如下所示。

```
func (a *applierV3backend) Put(txn mvcc.TxnWrite, p *pb.PutRequest)
    (resp *pb.PutResponse, err error) {
  resp = &pb.PutResponse{} // 该方法的最终返回值
  resp.Header = &pb.ResponseHeader{}
  val, leaseID := p.Value, lease.LeaseID(p.Lease)
  if txn == nil { // 如果传入的 txn 事务为空，则开启新事务
    // 通过 lessor.Lookup()方法查找 PutRequest 携带的 LeaseID 是否存在，如果不存在，则返回错误信息
    txn = a.s.KV().Write()
    defer txn.End() // 方法结束时关闭该方法中开启的事务
  }

  var rr *mvcc.RangeResult
  if p.IgnoreValue || p.IgnoreLease || p.PrevKv {
    // 如果设置了 IgnoreValue、IgnoreLease 或是 PreKv，则需要当前的键值对信息，错误处理(略)
    rr, err = txn.Range(p.Key, nil, mvcc.RangeOptions{})
  }
  // 如果设置了 IgnoreValue 或 IgnoreLease，则检测当前键值对是否存在，如果不存在则返回错误(略)
  if p.IgnoreValue { // 如果设置了 IgnoreValue，则使用当前的 Value 值
    val = rr.KVs[0].Value
  }
  if p.IgnoreLease { // 同理，如果设置了 IgnoreLease，则不会更新键值对绑定的 Lease
    leaseID = lease.LeaseID(rr.KVs[0].Lease)
```

```
    }
    if p.PrevKv { // 如果设置了 PreKv，则在 PutResponse 中返回更新前的键值对信息
        if rr != nil && len(rr.KVs) != 0 {
            resp.PrevKv = &rr.KVs[0]
        }
    }
    // 调用 Put()方法完成更新操作，在 PutResponse.Header 中记录 revision 信息
    resp.Header.Revision = txn.Put(p.Key, val, leaseID)
    return resp, nil
}
```

➢ Range

在 RangeRequest 中封装 applierV3backend.Range()方法进行查询时用到的相关参数，其中各字段的含义如下所示。

- Key（[]byte 类型）：查询的起始 Key 值。
- RangeEnd（[]byte 类型）：查询的结束 Key 值。
- Limit（int64 类型）：查询返回键值对的限制数，如果将其设置为 0，则表示无限制。
- Revision（int64 类型）：查询在指定的 Revision 时的键值对信息。
- SortOrder（RangeRequest_SortOrder 类型）：查询结果集的排序方式（0 表示不进行排序，1 表示正序排序，2 表示反向排序）。
- SortTarget（RangeRequest_SortTarget 类型）：查询结果集按照哪个字段进行排序（0～4 依次表示按照 Key、Version、Create、Mod、Value 进行排序）。
- Serializable（bool 类型）：默认的读操作都是前面介绍的 Linearizable Read，如果不需要 Linearizable Read 这种一致性，则可以将该字段设置为 true，实现 Local Read。
- KeysOnly（bool 类型）：如果将该字段设置成 true，则查询结果集中只包含键值对中的 Key 值，不包含 Value 值。
- CountOnly（bool 类型）：如果将该字段设置为 true，则查询结果只返回符合查询条件的键值对个数，而不包含键值对的其他信息。
- MinModRevision、MaxModRevision（int64 类型）：指定查询结果的 revision 范围。
- MinCreateRevision、MaxCreateRevision（int64 类型）：指定查询结果的 CreateRevision 范围。

applierV3backend.Range()方法会根据 RangeRequest 中指定的条件进行查询，并且对结果集进行过滤，其具体实现如下：

```
func (a *applierV3backend) Range(txn mvcc.TxnRead, r *pb.RangeRequest)
      (*pb.RangeResponse, error) {
  resp := &pb.RangeResponse{} // 该方法的返回值
  resp.Header = &pb.ResponseHeader{}
  // 与 Put()方法类似,如果当前只读事务为空,则开启新的只读事务,并在该 Range 查询结束后关闭此事务(略)
  limit := r.Limit
  if r.SortOrder != pb.RangeRequest_NONE || r.MinModRevision != 0
     || r.MaxModRevision != 0 || r.MinCreateRevision != 0 || r.MaxCreateRevision != 0 {
    limit = 0    // 如果指定了上述查询条件,则先查询全部符合条件的 KV,然后进行 limit 的过滤
  }
  // 修正当前的 limit 值(略)
  ro := mvcc.RangeOptions{  // 创建 RangeOptions,其中封装了 limit、revision 等查询条件
    Limit: limit,
    Rev:   r.Revision,
    Count: r.CountOnly,
  }
  // 调用 TxnRead.Range()方法进行查询,该方法的具体实现在前面已经详细分析过了,这里不再赘述
  rr, err := txn.Range(r.Key, r.RangeEnd, ro)

  if r.MaxModRevision != 0 { // 根据 MaxModRevision 对查询结果进行过滤
    f := func(kv *mvccpb.KeyValue) bool { return kv.ModRevision > r.MaxModRevision }
    pruneKVs(rr, f) // pruneKVs()函数中会根据上面定义的回调函数的返回值进行过滤
  }
  // 根据 MinModRevision、MaxCreateRevision、MinCreateRevision 对结果集进行过滤,与根
  // 据 MaxModRevision 对结果集进行过滤的实现基本类似,这里不再展开描述
  // 根据 SortOrder 和 SortTarget 对结果集进行排序(略)
  if r.Limit > 0 && len(rr.KVs) > int(r.Limit) { // 根据 limit 对结果集进行过滤
    rr.KVs = rr.KVs[:r.Limit]
    resp.More = true
  }
  // 将查询结果集中的键值对数据填充到 RangeResponse 中,并返回
  resp.Header.Revision = rr.Rev
  resp.Count = int64(rr.Count) // 此次返回的键值对个数
  for i := range rr.KVs {
    if r.KeysOnly {  // 只返回 Key 值
      rr.KVs[i].Value = nil
    }
```

```
        resp.Kvs = append(resp.Kvs, &rr.KVs[i])
    }
    return resp, nil
}
```

➢ DeleteRange

在 DeleteRangeRequest 中封装了 applierV3backend.DeleteRange()方法删除键值对时使用到的参数，其中各个字段的含义如下所示。

- **Key（[]byte 类型）**：删除的起始 Key 值。
- **RangeEnd（[]byte 类型）**：删除的结束 Key 值，如果未指定该值，则表示只删除上面的 Key 字段指定的键值对。
- **PrevKv（bool 类型）**：如果该字段设置为 true，则会返回删除前的键值对信息。

applierV3backend.DeleteRange()方法会根据 DeleteRangeRequest 中的条件删除指定的键值对信息，具体实现如下：

```
func (a *applierV3backend) DeleteRange(txn mvcc.TxnWrite, dr *pb.DeleteRangeRequest)
        (*pb.DeleteRangeResponse, error) {
    resp := &pb.DeleteRangeResponse{} // 返回值
    resp.Header = &pb.ResponseHeader{}
    // 与 Put()方法类似，如果当前只读事务为空，则开启新的只读事务，并在该 Range 查询结束后关闭此事务(略)
    // 修正 DeleteRangeRequest.RangeEnd 字段值(略)
    if dr.PrevKv { // 如果 PrevKv 设置为 true，则先查询指定范围的键值对，并封装到返回值中
        rr, err := txn.Range(dr.Key, dr.RangeEnd, mvcc.RangeOptions{})
        if rr != nil {
            for i := range rr.KVs {
                resp.PrevKvs = append(resp.PrevKvs, &rr.KVs[i])
            }
        }
    }
    // 调用 DeleteRange()方法删除指定范围的键值对，具体实现在前面已经分析过了，这里不再展开
    resp.Deleted, resp.Header.Revision = txn.DeleteRange(dr.Key, dr.RangeEnd)
    return resp, nil
}
```

➢ Txn

上面介绍的方法都是处理单个请求，applierV3backend.Txn()方法提供了批量操作的功能。

在 TxnRequest 中封装了批量操作需要的相关信息，其中有三个字段，分别是 Compare（[]*Compare 类型）、Success（[]*RequestOp 类型）和 Failure（[]*RequestOp 类型），每个字段中都封装了多个操作。在 Txn()方法中，首先会执行该字段中指定的比较操作，如果比较操作都返回 true，则执行 Success 字段中记录的操作，反之，则执行 Failure 字段中的操作。

applierV3backend.Txn()方法的具体实现如下：

```
func (a *applierV3backend) Txn(rt *pb.TxnRequest) (*pb.TxnResponse, error) {
  // 同时检测 TxnRequest.Success 和 Failure 中封装的操作,如果两者的全部操作都是读操作,则返回 false
  isWrite := !isTxnReadonly(rt)
  // 开启只读事务。NewReadOnlyTxnWrite()函数返回一个 txnReadWrite 实例，该实例虽然实现了
  // TxnWrite 接口，但是其底层依赖只读事务完成读操作，其写操作(Put、DeleteRange 等)都是不可用的
  txn := mvcc.NewReadOnlyTxnWrite(a.s.KV().Read())
  // 执行 TxnRequest.Compare 中指定的比较操作，其具体实现在下面展开描述
  reqs, ok := a.compareToOps(txn, rt)
  // 检测 reqs 中封装的操作是否合法(略)
  resps := make([]*pb.ResponseOp, len(reqs))
  txnResp := &pb.TxnResponse{
    Responses: resps,
    Succeeded: ok,
    Header:    &pb.ResponseHeader{},
  }
  if isWrite { // 如果 TxnRequest 中包含写操作，则结束当前只读事务，开启读写事务
    txn.End()
    txn = a.s.KV().Write()
  }
  for i := range reqs {// 执行真正的操作，并记录相应的返回值
    resps[i] = a.applyUnion(txn, reqs[i])
  }
  rev := txn.Rev() // 获取当前的 revision
  if len(txn.Changes()) != 0 {
    rev++ // 如果此次事务中有更新操作，则递增 rev
  }
  txn.End() // 提交事务
  txnResp.Header.Revision = rev // 在返回值中记录最新的 revision
  return txnResp, nil
}
```

在 compareToOps()函数中，会遍历 TxnRequest.Compare 字段并逐个调用 applyCompare()函数完成检测，如果其中任意一次比较返回 false，则 compareToOps()函数返回 TxnRequest.Failure，反之，则返回 TxnRequest.Success。下面简单分析 applyCompare()函数的实现，其中还会介绍 Compare 中各字段的含义：

```
func applyCompare(rv mvcc.ReadView, c *pb.Compare) bool {
  rr, err := rv.Range(c.Key, nil, mvcc.RangeOptions{}) // 查询指定键值对，错误处理(略)
  var ckv mvccpb.KeyValue
  if len(rr.KVs) != 0 {
    ckv = rr.KVs[0]
  } else {
    // 未查询到键值对的处理(略)
  }

  var result int
  // 在 Compare.Target 指定了比较 KLeyValue 中的哪个字段，在其 TargetUnion 字段中记录了对应的值
  switch c.Target {
  case pb.Compare_VALUE:
    tv, _ := c.TargetUnion.(*pb.Compare_Value)
    if tv != nil {
      result = bytes.Compare(ckv.Value, tv.Value) // 比较 Value 值
    }
  case pb.Compare_CREATE: // 其他字段的比较(略)
  case pb.Compare_MOD:
  case pb.Compare_VERSION:
  }

  switch c.Result { // 在 Compare.Result 字段中记录了此次比较成功的条件
  case pb.Compare_EQUAL:
    return result == 0
  case pb.Compare_NOT_EQUAL:
    return result != 0
  // 省略“>”和“<”的相关代码
  }
  return true
}
```

最后看一下 applierV3backend.applyUnion()方法，其中会根据传入的操作类型进行分类处理，

这里的操作有 Range、Put 和 DeleteRange 三种类型。applierV3backend.applyUnion()方法的具体实现如下：

```
func (a *applierV3backend) applyUnion(txn mvcc.TxnWrite, union *pb.RequestOp)
    *pb.ResponseOp {
  switch tv := union.Request.(type) { // 获取此操作的类型
  case *pb.RequestOp_RequestRange:
    if tv.RequestRange != nil {
      resp, err := a.Range(txn, tv.RequestRange) // 调用 applierV3backend.Range()方法
      return &pb.ResponseOp{Response: &pb.ResponseOp_ResponseRange{ResponseRange:
resp}}
    }
  case *pb.RequestOp_RequestPut: // 同理，调用 applierV3backend.Put()方法进行处理(略)
  case *pb.RequestOp_RequestDeleteRange: // 同理，调用 applierV3backend.Put()方法进行
                                         // 处理(略)
  default:
    return nil
  }
  return nil
}
```

➢ Compact

applierV3backend.Compact()方法是用来处理 CompactionRequest 请求的，其底层会调用前面介绍的 v3 存储的 Compact()方法完成压缩，具体实现如下：

```
func (a *applierV3backend) Compaction(compaction *pb.CompactionRequest)
    (*pb.CompactionResponse, <-chan struct{}, error) {
  resp := &pb.CompactionResponse{}
  resp.Header = &pb.ResponseHeader{}
  ch, err := a.s.KV().Compact(compaction.Revision)   // 完成压缩操作
  // 主要是为了获取最新的 revision
  rr, _ := a.s.KV().Range([]byte("compaction"), nil, mvcc.RangeOptions{})
  resp.Header.Revision = rr.Rev
  return resp, ch, err
}
```

applierV3backend 中剩余的其他方法都是调用 EtcdServer 中的相应字段的相应方法完成的。

- applierV3backend.Lease*()方法委托给了 EtcdServer.lessor；
- applierV3backend.Alarm()方法委托给了 EtcdServer.alarmStore；
- applierV3backend.Auth*()、User*()方法和 Role*()方法都委托给了 EtcdServer.authStore。

这些方法的具体实现比较简单，后面也会简单介绍，这里就不再赘述。

2. quotaApplierV3 和 applierV3Capped

quotaApplierV3 是 applierV3 接口的另一实现，也是一个装饰器，在原有 applierV3 实现的基础上添加了限流功能。在 quatoApplierV3 中除了内嵌了 applierV3 接口，还封装了一个 Quota 实例。Quota 接口定义了实现限流功能的核心，其具体定义如下所示。

```
type Quota interface {
    // 检测此次请求是否能通过限流，即当前请求执行后，未达到系统负载的上限，也就未触发限流
    Available(req interface{}) bool
    // 计算此次请求所产生的负载，在后面介绍具体实现时会看到，该方法主要在 Available()方法中调用
    Cost(req interface{}) int
    // 当前系统所能支持的剩余负载量
    Remaining() int64
}
```

结构体 backendQuota 是 Quota 接口的实现之一，它主要用于限制底层 BoltDB 中的数据量，其中封装了当前节点的 EtcdServer 实例（即 s 字段）和 BoltDB 数据量的上限值（maxBackendBytes 字段）。

首先看一下 NewBackendQuota()函数，其中会根据传入的参数初始化相应的 Quota 实例，其具体实现如下：

```
func NewBackendQuota(s *EtcdServer) Quota {
    if s.Cfg.QuotaBackendBytes < 0 {
        // 创建 passthroughQuota 实例，它也是 Quota 接口的实现之一，但它并没有实现限流的功能，
        // 其 Available()方法始终返回 true，Cost()方法始终返回 false
        return &passthroughQuota{}
    }
    if s.Cfg.QuotaBackendBytes == 0 {
        return &backendQuota{s, DefaultQuotaBytes}  // 创建 backendQuota 实例
    }
    return &backendQuota{s, s.Cfg.QuotaBackendBytes} // 创建 backendQuota 实例
}
```

backendQuota.Available()方法中会将当前 BoltDB 中的数据量、此次请求的数据量之和与上限阈值进行比较，从而决定此次请求是否触发限流，具体实现如下所示。

```
func (b *backendQuota) Available(v interface{}) bool {
  return b.s.Backend().Size()+int64(b.Cost(v)) < b.maxBackendBytes
}
```

请求的数据量是通过 Cost()方法得到的，在其 Cost()方法实现中，会根据请求的类型进行分类计算，具体实现如下所示。

```
// Cost()方法中根据请求的类型行分类处理
func (b *backendQuota) Cost(v interface{}) int {
  switch r := v.(type) {
  case *pb.PutRequest: // 这里
    return costPut(r)
  case *pb.TxnRequest:
    return costTxn(r)
  case *pb.LeaseGrantRequest: // Lease 相关的请求，后面详细介绍
    return leaseOverhead
  default: // 未知请求
  }
}
```

costPut()方法会计算请求的 Key 值、Value 值和相关元数据的字节数之和，costTxn()方法则是调用 costPut()方法计算 TxnRequest.Success 和 TxnRequest.Failure，它们的具体实现比较简单，如下所示。

```
func costPut(r *pb.PutRequest) int { return kvOverhead + len(r.Key) + len(r.Value) }

func costTxn(r *pb.TxnRequest) int {
  sizeSuccess := 0
  for _, u := range r.Success {
    sizeSuccess += costTxnReq(u)
  }
  sizeFailure := 0
  // 与计算 r.Success 同理，计算 r.Failure 的数据量
  if sizeFailure > sizeSuccess {
    return sizeFailure
```

```
    }
    return sizeSuccess
}

// costTxnReq()方法直接调用了 costPut()方法，不再粘贴代码
```

在 quatoApplierV3 中，为写操作（例如，Put()方法、Txn()方法等）添加了限流的功能，这里以 Put()方法为例进行介绍，其他的方法实现类似，留给读者分析。在 quatoApplierV3.Put()方法中，首先会调用 Quota.Available()方法检测此次请求是否会触发限流，如果未触发，则委托给底层的 applierV3 实现完成真正的 Put 操作，具体实现如下所示。

```
func (a *quotaApplierV3) Put(txn mvcc.TxnWrite, p *pb.PutRequest) (*pb.PutResponse,
error) {
    ok := a.q.Available(p) // 检测是否触发限流
    resp, err := a.applierV3.Put(txn, p) // 完成真正的 Put 操作
    if err == nil && !ok {
        err = ErrNoSpace // 注意返回的错误信息
    }
    return resp, err
}
```

注意，在这里触发限流之后会返回 ErrNoSpace 错误。读者可以回顾一下前面介绍的 EtcdServer.applyEntryNormal()方法，当收到 applierV3.Apply()方法返回的 ErrNoSpace 错误之后，会向集群其他节点发送 AlarmRequest 请求。当该请求经过 Raft 协议提交之后，集群中各个节点在应用该请求时，就会新建一个 applierV3Capped 实例，并将当前 EtcdServer.applyV3 字段指向该实例。

applierV3Capped 是 applierV3 接口的另一实现，其中所有的写操作都会返回 ErrNoSpace 错误，从而保证底层的 BoltDB 库中的数据量不再增加。applierV3Capped 的实现比较简单，这里不再展开介绍。

3. authApplierV3

最后看一下 applierV3 接口的另一个实现——authApplierV3，它也是一个装饰器，其中内嵌了其他 applierV3 实现，并在其基础上添加了权限控制的功能。authApplierV3 中各个字段的含义如下所示。

- as（auth.AuthStore 类型）：AuthStore 接口中定义与权限管理相关操作，后面会详细介绍其具体实现。

- authInfo (auth.AuthInfo 类型): 在处理每个请求时，都会使用该字段记录该请求头中的权限信息，AuthInfo 结构体中有两个字段，分别记录了请求头中携带的 Username 和 Revision 信息。

在 authApplierV3 实现中重写了 Apply()方法，其中首先记录请求头中携带的权限信息，然后对此次请求是否需要 Admin 权限进行检测，最后调用底层 applierV3 实现的 Apply()方法完成请求的分发。authApplierV3.Apply()方法的具体实现如下：

```
func (aa *authApplierV3) Apply(r *pb.InternalRaftRequest) *applyResult {
  if r.Header != nil { // 将请求头中的 Username 和 Revision 记录到 authApplierV3.authInfo 中
    aa.authInfo.Username = r.Header.Username
    aa.authInfo.Revision = r.Header.AuthRevision
  }
  // 检测该请求是否需要 Admin 权限，其中 AuthEnable、AuthDisable、AuthUser*、AuthRole*等请
  // 求，都是需要 Admin 权限的
  if needAdminPermission(r) {
    if err := aa.as.IsAdminPermitted(&aa.authInfo); err != nil ... // 检测 Admin 权限
  }
  ret := aa.applierV3.Apply(r) // 调用底层 applierV3 实现的 Apply()方法完成请求分发
  aa.authInfo.Username = "" // 清空 authApplierV3.authInfo
  aa.authInfo.Revision = 0
  return ret
}
```

在 authApplierV3 中重写了 Put()、Range()和 Txn()等方法，并在其中添加了权限检查的相关逻辑，这些方法的实现大致类似，这里以 Put()方法为例进行分析，其他方法留给读者分析。

```
func (aa *authApplierV3) Put(txn mvcc.TxnWrite, r *pb.PutRequest) (*pb.PutResponse,
error) {
  if err := aa.as.IsPutPermitted(&aa.authInfo, r.Key); err != nil ... // 检测 Put 权限
  if r.PrevKv { // 如果需要返回更新前的键值对信息，则需要用 Range 权限
    err := aa.as.IsRangePermitted(&aa.authInfo, r.Key, nil) // 省略错误处理
  }
  return aa.applierV3.Put(txn, r) // 调用底层的 applierV3 实现，完成 Put 操作
}
```

至此，applierV3 接口的实现就全部介绍完了。最后，我们看一下 EtcdServer 中相关字段，从前面分析的 EtcdServer 初始化过程（NewServer()函数）中可以看到，applyV3Base 字段实际

上是指向了本节介绍的 applierV3backend 实例，applyV3 字段指向了 authApplierV3 实例，而该 authApplierV3 实例底层封装了 quatoApplierV3 实例。

7.5 AlarmStore

在本章前面介绍 quotaApplierV3 时，分析了 AlarmStore 与 quotaApplierV3 配合实现限流功能的具体实现。本节将详细介绍 AlarmStore 的具体实现，首先看一下 AlarmStore 中各个字段的含义。

- types（map[pb.AlarmType]alarmSet 类型）：在该 map 字段中记录了每种 AlarmType 对应的 AlarmMember 实例。AlarmType 现在只有 AlarmType_NONE 和 AlarmType_NOSPACE 两种类型。alarmSet 类型实际上是 map[types.ID]*pb.AlarmMember 类型，其中记录了节点 ID 与 AlarmMember 之间的映射关系。
- bg（BackendGetter 类型）：BackendGetter 接口用于返回该 AlarmStore 实例使用的存储。EtcdServer 就是 BackendGetter 接口的实现之一，返回的就是其底层使用的 backend 实例。

在前面介绍 quotaApplierV3 时我们提到，当首次触发限流时会创建 AlarmRequest 请求并封装成 MsgProp 消息发送到集群当中，在 AlarmRequest 中封装的信息如下所示。

- Action（AlarmRequest_AlarmAction 类型）：该字段有 AlarmRequest_GET、AlarmRequest_ACTIVATE 和 AlarmRequest_DEACTIVATE 三种值，分别对应使用 AlarmStore.Get()、Activate()和 Deactivate()方法三个方法进行该 AlarmRequest 请求。
- MemberID（uint64 类型）：记录发出该 AlarmRequest 请求的节点 ID。
- Alarm（AlarmType 类型）：记录该 AlarmRequest 请求相关的 Alarm 类型。

当 AlarmRequest 请求经过 Raft 协议提交之后，会调用 applierV3backend.Alarm()方法应用相应的 Entry 记录，该方法会根据 AlarmRequest.Action 字段分类处理，具体实现如下：

```
func (a *applierV3backend) Alarm(ar *pb.AlarmRequest) (*pb.AlarmResponse, error) {
  resp := &pb.AlarmResponse{}
  oldCount := len(a.s.alarmStore.Get(ar.Alarm))
  switch ar.Action {
  case pb.AlarmRequest_GET: // 调用 AlarmStore.Get()方法处理 AlarmRequest_GET 类型的
                            // Action
    resp.Alarms = a.s.alarmStore.Get(ar.Alarm)
  // 调用 AlarmStore.Activate()方法处理 AlarmRequest_ACTIVATE 类型的 Action
  case pb.AlarmRequest_ACTIVATE:
```

```
        m := a.s.alarmStore.Activate(types.ID(ar.MemberID), ar.Alarm)
        if m == nil {
            break
        }
        resp.Alarms = append(resp.Alarms, m)
        activated := oldCount == 0 && len(a.s.alarmStore.Get(m.Alarm)) == 1
        if !activated {
            break
        }

        switch m.Alarm {
        case pb.AlarmType_NOSPACE:
            a.s.applyV3 = newApplierV3Capped(a)
        }
    // 调用 AlarmStore.Activate()方法处理 AlarmRequest_DEACTIVATE 类型的 Action
    case pb.AlarmRequest_DEACTIVATE:
        m := a.s.alarmStore.Deactivate(types.ID(ar.MemberID), ar.Alarm)
        if m == nil {
            break
        }
        resp.Alarms = append(resp.Alarms, m)
        deactivated := oldCount > 0 && len(a.s.alarmStore.Get(ar.Alarm)) == 0
        if !deactivated {
            break
        }

        switch m.Alarm {
        case pb.AlarmType_NOSPACE:
            a.s.applyV3 = a.s.newApplierV3()
        }
    default:
        return nil, nil
    }
    return resp, nil
}
```

下面看一下 AlarmStore.Get()方法，如果传入的参数为 AlarmType_NONE，则返回该 AlarmStore 实例中的全部 AlarmMember 实例，具体实现如下：

仅供非商业用途或交流学习使用

```
func (a *AlarmStore) Get(at pb.AlarmType) (ret []*pb.AlarmMember) {
  a.mu.Lock()
  defer a.mu.Unlock()
  if at == pb.AlarmType_NONE { // 针对 AlarmType_NONE 类型的 AlarmType 处理
    for _, t := range a.types { // 遍历 types 字段中全部的 AlarmMember 返回
      for _, m := range t {
        ret = append(ret, m)
      }
    }
    return ret
  }
  for _, m := range a.types[at] { // 只返回指定类型的 AlarmMember 实例
    ret = append(ret, m)
  }
  return ret
}
```

AlarmStore.Activate()方法负责新建 AlarmMember 实例，并将其记录到 AlarmStore.types 字段和底层存储中，具体实现如下：

```
func (a *AlarmStore) Activate(id types.ID, at pb.AlarmType) *pb.AlarmMember {
  newAlarm := &pb.AlarmMember{MemberID: uint64(id), Alarm: at} // 新建 AlarmMember
                                                               // 实例
  if m := a.addToMap(newAlarm); m != newAlarm { // 将 AlarmMember 添加到 types 字段
    return m
  }
  v, err := newAlarm.Marshal() // 将 AlarmMember 实例序列化
  b := a.bg.Backend() // 获取底层的存储
  b.BatchTx().Lock()
  b.BatchTx().UnsafePut(alarmBucketName, v, nil)  // 将 AlarmMember 持久化到底层存储中
  b.BatchTx().Unlock()
  return newAlarm
}
```

AlarmStore.Deactivate()方法负责从 types 字段和底层存储中删除指定的 AlarmMember 实例，具体实现如下：

```
func (a *AlarmStore) Deactivate(id types.ID, at pb.AlarmType) *pb.AlarmMember {
```

仅供非商业用途或交流学习使用

```
    t := a.types[at] // 根据 AlarmType 类型查找 alarmSet
    if t == nil {
        t = make(alarmSet)
        a.types[at] = t
    }
    m := t[id] // 根据 id 查找 AlarmMember 实例
    if m == nil {
        return nil
    }
    delete(t, id) // 从 types 字段中删除指定的 AlarmMember 实例
    v, err := m.Marshal()
    b := a.bg.Backend()
    b.BatchTx().Lock()
    b.BatchTx().UnsafeDelete(alarmBucketName, v) // 从底层存储中删除指定的 AlarmMember 信息
    b.BatchTx().Unlock()
    return m
}
```

本章小结

本章主要介绍了 etcd-server 模块的具体实现。在 etcd-server 模块中，使用 RaftCluster 维护当前集群中的节点信息，任何节点信息的修改都会修改其中记录的 Member 信息，并持久化到底层存储中。etcd-server 模块中定义的 raftNode 是 etcd-server 模块与底层 etcd-raft 模块交互的桥梁，它主要负责处理 etcd-raft 模块返回的 Ready 实例，例如，调用 Transport 发送 Ready 实例中携带的待发送消息，将 Ready 实例中携带的待应用 Entry 记录返回给 EtcdServer 处理，等等。

结构体 EtcdServer 是 etcd-server 模块的核心，它会启动并协调多个后台 goroutine 的工作。在 EtcdServer 初始化时，会根据配置项初始化 etcd-raft 模块，其中涉及 WAL 日志文件的初始化、etcd v2 存储和 etcd v3 存储的初始化，以及网络层 RoundTripper 实例的初始化。初始化完成之后，EtcdServer 实例会注册集群通信使用的 Handler 实例，也会将当前节点的 URL 发布到集群当中。除此之外，EtcdServer 启动的多个后台 goroutine 会完成定期清理 WAL 日志文件，以及快照文件、处理 Linearizable Read 请求等功能。

最后，我们详细介绍了 ApplierV2 接口和 applierV3 接口的具体实现，它们是 EtcdServer 处理待应用 Entry 记录的核心。ApplierV2 底层依赖的是 etcd v2 存储，applierV3 接口的实现底层依赖的是 etcd v3 存储。在 applierV3 接口的实现中，applierV3backend 是最基础的实现，提供了

仅供非商业用途或交流学习使用

最基础的持久化和查询的功能；quotaApplierV3 和 applierV3Capped 首先都是装饰器，它们相互配合工作，在原来的 applierV3backend 之上实现限流的功能；authApplierV3 实现也是装饰器，在原有 applierV3 实现的基础上提供了权限控制的功能。

希望通过本章的介绍，读者可以了解 etcd-server 模块的工作原理和具体实现，了解 etcd-server 模块如何组装其他模块并协调它们的工作。

仅供非商业用途或交流学习使用



第 8 章 etcd 客户端详解

在 etcd v2 版本中，客户端采用的是 HTTP+JSON 的方式与 Server 端进行交互。在 etcd v3 版本中，客户端采用的是 GRPC 的方式与 Server 端进行交互。相比之下，Client v2 的交互方式更加简单快捷，我们甚至可以直接使用浏览器或 Post Man 等工具与 etcd v2 集群进行交互。Client v3 使用 GRPC 进行通信，底层的协议消息通过 Google Protocol Buffer 定义，它可以简化 RPC 客户端存根代码的生成和管理。另外，GRPC 在处理网络连接方面优势比较明显，GRPC 使用单一连接的 HTTP2，实现多路复用的 RPC 调用，而 HTTP+JSON 的客户端实现则必须为每个请求建立一个连接。

8.1 GRPC基础

在开始介绍 etcd 对 GRPC 的应用之前，我们先来介绍一下 GRPC 的基础使用方式。在本节中，我们将简单介绍在 Golang 中使用 GRPC 的基本流程，并介绍 GRPC 自带的 route_guide 示例。使用 GRPC 编写一个服务分为如下三步：

（1）编写一个“.proto”文件，在该“.proto”文件中定义相关服务。

（2）用 Protocol Buffer 编译器生成服务端和客户端的存根代码。

（3）使用 GRPC 的 Go API 为服务实现一个简单的客户端和服务器。

8.1.1 定义proto文件

这里我们以 GRPC 自带的 route_guide 示例介绍 GRPC 的基本使用方法。首先，我们需要获

仅供非商业用途或交流学习使用

取 GRPC 的代码，具体命令如下：

```
go get google.golang.org/grpc
```

之后，我们可以跳转到 route_guide 实例的目录中，具体命令如下：

```
cd $GOPATH/src/google.golang.org/grpc/examples/route_guide
```

在 route_guide 目录下，我们可以看到一个名为“route_guide.proto”的文件，在其中定义 RouteGuide 服务，具体内容如下所示。

```
// 省略 syntax 和 option
package routeguide;

service RouteGuide { // 定义 RouteGuide 服务
  // 定义一个普通的 RPC 方法。客户端调用 GetFeature()方法时，会像调用本地方法一样阻塞等待。客户端
  // 会通过网络将 Point 序列化后发送到服务端，并等待服务端响应。当客户端得到服务端响应之后，其调
  // 用的 GetFeature()方法才会返回
  rpc GetFeature(Point) returns (Feature) {}

  // 定义一个流式 RPC。在调用 ListFeatures()方法时，客户端会拿到一个流去读取返回的消息序列。客户
  // 端循环读取返回的流，直到里面没有任何消息可读。从例子中可以看出，通过在响应类型前插入 stream
  // 关键字，可以指定一个服务器端的流方法
  rpc ListFeatures(Rectangle) returns (stream Feature) {}

  // 定义一个客户端流式 RPC。在调用 RecordRoute()方法时，客户端写入一个消息序列并将其发送到服务
  // 器，同样也是使用流。一旦客户端完成写入消息，就会开始等待服务器完成读取返回响应
  rpc RecordRoute(stream Point) returns (RouteSummary) {}

  // 定义一个双向流式 RPC。客户端和服务端都使用读写流的方式去发送一个消息序列，两个流独立操作，因
  // 此客户端和服务器可以以任意喜欢的顺序读写。例如，服务器可以在写入响应前等待接收所有的客户端消
  // 息，或者可以交替地读取和写入消息，或者是其他读写的组合
  rpc RouteChat(stream RouteNote) returns (stream RouteNote) {}
}
```

在 route_guide.proto 文件中，除了定义 RouteGuide 服务，还定义了该服务中使用到的消息，如下所示。

```
message Point { // 定义 Point 消息，其中封装了两个 int 字段，分别是 latitude 和 longitude
```

仅供非商业用途或交流学习使用

```
  int32 latitude = 1;
  int32 longitude = 2;
}

message Rectangle { // 定义 Rectangle 消息，其中封装了两个 Point，分别是 lo 和 hi
  Point lo = 1;
  Point hi = 2;
}
// 其他消息的定义类似，这里不再赘述
```

完成“.proto”文件的定义之后，我们可以通过 protoc（Protocol Buffer 的编译器）和一个特殊的 GRPC-Go 插件来生成 GRPC 客户端和服务器端的接口。在 Mac 系统上安装 protoc 的步骤如下，其他系统的读者可以查阅相关资料进行学习。

（1）从 Protocol Buffers 官方网站（https://github.com/google/protobuf/releases）下载最新的 Protocol Buffers 压缩包，并解压缩。

（2）将命令行跳转到解压后的目录中，并执行“./configure”命令。

（3）依次执行“make”命令、“make check”命令、“sudo make install”命令。

（4）通过“protoc –version”命令检测安装是否成功。

完成 protoc 的安装之后，我们回到前面介绍的 route_guide 目录下并执行“protoc --go_out=plugins=grpc:. route_guide.proto”命令，即可在当前目录中得到一个名为“route_guide.pb.go”文件。在该文件中包括如下内容：

- 所有用于填充、序列化和获取上述定义的请求和响应消息类型的 Protocol Buffers 代码。
- 一个为客户端调用定义的接口（RouteGuideClient 接口）。
- 一个为服务端定义的接口，其中定义了前面介绍的 RouteGuide 服务中的全部方法（RouteGuideServer 接口）。

8.1.2　服务端

接下来，我们需要创建一个 RouteGuide 服务器。首先需要做的就是实现前面生成的 RouteGuideServer 接口。在 route_guide 示例的 server/server.go 文件中，我们可以看到 routeGuideServer 结构体，它就是我们需要的 RouteGuideServer 接口实现。这里简单介绍一下其中的几个方法实现。

首先是 routeGuideServer.GetFeature()方法，该方法的第一个参数是 RPC 的上下文对象，第

仅供非商业用途或交流学习使用

二个参数是客户端传递的 Point 消息，返回了一个包含响应信息的 Feature 消息和一个错误信息。该方法的大致实现如下：

```
func (s *routeGuideServer) GetFeature(ctx context.Context, point *pb.Point)
    (*pb.Feature, error) {
  // 这里我们不关注具体的业务逻辑(略)
  return &pb.Feature{Location: point}, nil
}
```

下面再看一下 routeGuideServer.ListFeatures()方法，它是一个流式 RPC，可以返回多个 Feature 消息给客户端。该方法的第一个参数是客户端发送的 Rectangle 消息；第二个参数是 RouteGuide_ListFeaturesServer 实例，它是用来写入响应消息的，在该方法中我们可以创建任意个数的 Feature 实例，然后通过 Send()方法把它们写入 RouteGuide_ListFeaturesServer 中。在该方法的最后，我们返回了一个 nil 告诉 GRPC 响应的写入已经完成。如果在调用过程中发生任何错误，我们可以返回一个非 nil 的错误信息通知 GRPC 层。routeGuideServer.ListFeatures()方法的具体实现如下：

```
func (s *routeGuideServer) ListFeatures(rect *pb.Rectangle,
    stream pb.RouteGuide_ListFeaturesServer) error {
  for _, feature := range s.savedFeatures {
    if inRange(feature.Location, rect) {
      if err := stream.Send(feature); err != nil { // 向客户端返回新的 Feature 实例
        return err
      }
    }
  }
  return nil
}
```

接下来看一下 routeGuideServer.RecordRoute()方法，这个方法只有一个 RouteGuide_RecordRouteServer 流作为其参数，服务端可以用它来读写消息。服务端可以用其 RouteGuide_RecordRouteServer.Recv()方法接收客户端的消息，也可以用其 SendAndClose()方法返回它的单个响应。RecordRoute()方法的具体实现如下：

```
func (s *routeGuideServer) RecordRoute(stream pb.RouteGuide_RecordRouteServer) error {
  for {
    point, err := stream.Recv()// 调用 Recv()方法读取客户端发送的消息
    if err == io.EOF {
```

仅供非商业用途或交流学习使用

```
            return stream.SendAndClose(&pb.RouteSummary{  // 通过 SendAndClose()方法返回响应
                ... ...
            })
        }
    }
}
```

最后看一下 routeGuideServer.RouteChat()方法，这个方法与上面的介绍的 RecordRoute()方法很类似，该方法的唯一参数是 RouteGuide_RouteChatServer 流，它可以用来读写消息。与 RecordRoute()方法的区别在于，当客户端还在往其中写入消息时，服务端同时可以通过 Send()方法返回消息，这样就可以返回多个消息。RouteChat()方法的具体实现如下：

```
func (s *routeGuideServer) RouteChat(stream pb.RouteGuide_RouteChatServer) error {
    for {
        in, err := stream.Recv()// 调用 Recv()方法读取客户端发送的消息
        // 这里我们不关注具体的业务逻辑(略)
        for _, note := range s.routeNotes[key] {
            if err := stream.Send(note); err != nil { // 通过 SendAndClose()方法返回响应
                return err
            }
        }
    }
}
```

了解完 routeGuideServer 结构体的实现，为服务端生成代码之后，我们还需要启动一个 GRPC 服务器，这样客户端才可以使用上述服务。在 route_guide 示例中的 server/server.go 文件中，我们可以看到 main()函数，其大致实现如下所示。

```
func main() {
    flag.Parse()
    // 指定监听的地址和端口，该地址即为服务地址
    lis, err := net.Listen("tcp", fmt.Sprintf("localhost:%d", *port))
    // 省略 TLS
    // 使用 grpc.NewServer()创建 GRPC 服务器的一个实例
    grpcServer := grpc.NewServer(opts...)
    // 在 GRPC 服务器注册上面介绍的 RouteGuideServer 服务实现
    pb.RegisterRouteGuideServer(grpcServer, newServer())
    // 调用 Serve()方法实现阻塞等待
```

仅供非商业用途或交流学习使用

```
    grpcServer.Serve(lis)
}
```

8.1.3 创建客户端

在 route_guide.pb.go 文件中，除了有 RouteGuideServer 接口，还有另一个供客户端使用的 RouteGuideClient 接口，并且提供了相应实现——结构体 routeGuideClient，其具体实现中涉及的 GRPC 使用方式与前面介绍的 routeGuideServer 有很多类似之处，这里不再展开描述。

接下来，我们开始搭建 RouteGuide 服务的 Go 客户端，在 route_guide 示例中的 client/client.go 文件中，我们可以看到其客户端的完整实现。这里重点关注其中的 main()方法，为了调用服务端方法，需要创建一个 GRPC channel 和服务端进行交互，具体实现如下：

```
func main() {
    flag.Parse()
    conn, err := grpc.Dial(*serverAddr, opts...)
    defer conn.Close()
    client := pb.NewRouteGuideClient(conn) // 创建 routeGuideClient 实例
    // 调用 RouteGuide 服务的 GetFeature()方法。如何调用 RouteGuide 服务的其他方法，这里不再展
    // 开介绍，感兴趣的读者可以参考 route_guide 实例中 client/client.go 的具体代码
    feature, err := client.GetFeature(context.Background(), point)
    log.Println(feature)
}
```

到此为止，GRPC 的基础使用和 route_guide 示例的实现就全部介绍完了。在后面介绍 etcd 的 Client v3 时，还会提到 GRPC 的相关内容。

8.2 Client v3

正如前面介绍的那样，在 etcd 3.2 中的客户端使用 GRPC 实现与服务端的交互。我们依旧按照上一节的介绍 GRPC 使用流程，先来看一下 proto 文件文件的定义，如图 8-1 所示，在 etcdserver/etcdserverpb 目录中的 rpc.proto 文件中里定义了六个服务。



图 8-1

下面先分析 KV 服务，其中定义了五个客户端最基本、最常用的 RPC 方法，分别是 Range()方法、Put()方法、Txn()方法、DeleteRange()方法和 Compact()方法，这些方法都是普通的 RPC 方法，没有使用前面介绍的流式操作，其具体定义如下所示。

```
service KV {
  rpc Range(RangeRequest) returns (RangeResponse) { // 查询键值对
     option (google.api.http) = {
       post: "/v3alpha/kv/range"
       body: "*"
    };
  }

  rpc Put(PutRequest) returns (PutResponse) { // 保存键值对
     option (google.api.http) = {
       post: "/v3alpha/kv/put"
       body: "*"
    };
  }

  rpc DeleteRange(DeleteRangeRequest) returns (DeleteRangeResponse) { // 删除键值对
     option (google.api.http) = {
       post: "/v3alpha/kv/deleterange"
       body: "*"
    };
  }

  rpc Txn(TxnRequest) returns (TxnResponse) {   // 事务操作
     option (google.api.http) = {
       post: "/v3alpha/kv/txn"
       body: "*"
    };
  }

  rpc Compact(CompactionRequest) returns (CompactionResponse) { // 压缩操作
     option (google.api.http) = {
       post: "/v3alpha/kv/compaction"
       body: "*"
    };
  }
}
```

在 rpc.proto 文件的同一目录下，有一个名为“rpc.pb.go”文件，其中就包含了通过 protoc 命令生成的服务端代码，定义了 KVServer 接口及该接口使用到的消息等内容。在 etcd-server 模块中提供了 KVServer 接口实现，如图 8-2 所示。

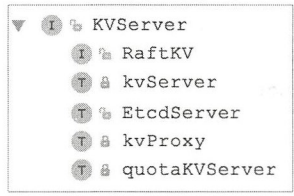


图 8-2

8.2.1 kvServer

接下来，我们深入分析结构体 kvServer 的具体实现，其中封装了两个字段，具体含义如下所示。

- hdr（header 类型）：用于填充响应消息的头信息。
- kv（etcdserver.RaftKV 类型）：如图 8-2 所示，该 RaftKV 接口继承了 KVServer 接口。在 NewKVServer()函数中我们可以看到，kvServer.kv 字段实际指向了前面介绍的 EtcdServer 实例。

kvServer 结构体处理请求的大致步骤如下：首先会对请求消息进行各方面的检查，检查完成之后会将所有的请求交给其内封装的 RaftKV 接口实现进行处理，待处理完成得到响应消息之后，会通过 header.fill()方法填充响应的头信息，最终将完整的响应消息返回给客户端，整个请求的处理流程结束。下面以 kvServer.Range()方法为例进行介绍，其具体实现如下所示。

```
func (s *kvServer) Range(ctx context.Context, r *pb.RangeRequest)
     (*pb.RangeResponse, error) {
  if err := checkRangeRequest(r); ... // 检测 RangeRequest 请求，错误处理(略)
  // 将 RangeRequest 请求委托给 kvServer 中封装的 RaftKV 实现进行处理，错误处理(略)
  resp, err := s.kv.Range(ctx, r)
  s.hdr.fill(resp.Header) // 调用 header.fill()填充响应的头信息
  return resp, nil
}
```

kvServer 中其他方法的实现与 Range()方法的处理流程基本一致，这里不再逐个展开描述了，

感兴趣的读者可以参考源码进行学习。

8.2.2　EtcdServer

前面介绍的 EtcdServer 结构体是 RaftKV 接口的实现之一，在结构体 kvServer 中 kv 字段就指向了 EtcdServer 示例。在本节中，我们将重点介绍 EtcdServer 对 RaftKV 接口的实现。

1. Range 方法

我们先来看一下 EtcdServer.Range()方法，它首先会将根据 RangeRequest 请求决定此次请求是 serializable 类型还是 linearizable 类型，这两种类型的读请求在前面也多次提到过，这里简单回顾一下，serializable read 请求会直接读取当前节点的数据并返回给客户端，它并不保证返回给客户端的数据是集群中最新的，例如，当前出现了网络分区，响应请求的节点是上一个 Term 的 Leader 节点。而 linearizable read 请求的处理过程会通过 Raft 协议保证返回给客户端最新数据，具体的过程在前面已经介绍过了，这里不再展开描述。之后，Range()会进行权限检查，检查通过之后会调用 EtcdServer.applyV3Base.Range()方法完成键值对的查询，最后，将键值对数据封装成 RangeResponse 消息返回给客户端。EtcdServer.Range()方法的具体实现如下所示。

```
    func (s *EtcdServer) Range(ctx context.Context, r *pb.RangeRequest)
(*pb.RangeResponse, error) {
        if !r.Serializable { // 判断此次请求是否为 linearizable read
            // 对于 Linearizable Read 来说，会在这里阻塞等待。关于 Linearizable Read 的处理，在前面
            // 上一章已经详细介绍过了，这里不再赘述。错误处理(略)
            err := s.linearizableReadNotify(ctx)
        }
        var resp *pb.RangeResponse
        var err error
        chk := func(ai *auth.AuthInfo) error {  // 用于检测 RangRequest 请求权限的回调函数
            return s.authStore.IsRangePermitted(ai, r.Key, r.RangeEnd)
        }
        get := func() { resp, err = s.applyV3Base.Range(nil, r) } // 真正查询键值对的回调函数
        // 下面重点来看 doSerialize()方法的处理流程，错误处理(略)
        if serr := s.doSerialize(ctx, chk, get); ...
        return resp, err
    }
```

在 EtcdServer.doSerialize()方法中，会使用 Range()方法中定义的回调函数，完成权限检测以及键值对数据的查询，其具体实现如下：

```
func (s *EtcdServer) doSerialize(ctx context.Context, chk func(*auth.AuthInfo) error,
    get func()) error {
  for {
    ai, err := s.AuthInfoFromCtx(ctx) // 获取权限相关的信息，错误处理(略)
    if ai == nil {
      ai = &auth.AuthInfo{}
    }
    if err = chk(ai); ... // 回调 chk()函数，通过 authStore 完成权限检测
    get() // 回调 get()函数，其中直接通过 applyV3Base 从底层存储中读取键值对数据
    //  若读取完成且没有发生权限修改，则结束此次读取，否则需要进行重试
    if ai.Revision == 0 || ai.Revision == s.authStore.Revision() {
      return nil
    }
  }
}
```

2. Put&DeleteRange

接下来，我们简单分析 EtcdServer.Put()和 DeleteRange()方法的具体实现。两者都会直接调用 EtcdServer.raftRequest()方法向集群发送 MsgProp 消息，并阻塞等待其中的 Entry 记录被应用。raftRequest()方法底层是调用 raftRequestOnce()方法实现的，如下：

```
func (s *EtcdServer) raftRequest(ctx context.Context, r pb.InternalRaftRequest)
    (proto.Message, error) {
  for {
    resp, err := s.raftRequestOnce(ctx, r)
    if err != auth.ErrAuthOldRevision { //  如果出现 ErrAuthOldRevision 错误，则重试
      return resp, err
    }
  }
}
```

raftRequestOnce()方法是通过调用 processInternalRaftRequestOnce()方法实现的，而 processInternalRaftRequestOnce()方法的具体实现在上一章中已经详细分析过了，这里不再展开介绍。

3. Txn

在介绍 applierV3backend.Txn()方法时也提到，在 TxnRequest 中可以封装多个操作，这些操

作将会批量执行。EtcdServer.Txn()方法就用来处理客户端发送的 TxnRequest，其中会根据 TxnRequest 中封装的操作类型进行分类处理。如果 TxnRequest 中封装的都是只读操作，则其处理流程与前面介绍 Range()方法类似；如果 TxnRequest 中包含写操作，则其处理流程与前面介绍的 Put()方法类似。Txn()方法的具体实现如下：

```
    func (s *EtcdServer) Txn(ctx context.Context, r *pb.TxnRequest) (*pb.TxnResponse,
error) {
        if isTxnReadonly(r) { // 检测 TxnRequest 中是否全部为只读操作
            // 如果 TxnRequest 中全部是只读操作，则检测这些操作是否全部为 serializable read
            if !isTxnSerializable(r) {
                // 如果存在 linearizable read，则调用 linearizableReadNotify()方法处理，并阻塞等
                // 待整个 linearizable read 处理流程结束。错误处理()略
                err := s.linearizableReadNotify(ctx)
            }
            var resp *pb.TxnResponse
            var err error
            // 权限检测回调函数(略)
            get := func() { resp, err = s.applyV3Base.Txn(r) } // 真正完成 TxnRequest 处理的
                                                               // 回调函数
            // 调用 doSerialize()方法，完成 Txn()处理，错误处理(略)
            if serr := s.doSerialize(ctx, chk, get);...
            return resp, err
        }
        // 如果 TxnRequest 中存在写操作，则需要调用 raftRequest()方法发送 MsgProp 消息，并等待其中
        // 的 Entry 记录被应用，错误处理(略)
        resp, err := s.raftRequest(ctx, pb.InternalRaftRequest{Txn: r})
        return resp.(*pb.TxnResponse), nil
    }
```

4. Compact

在前面的章节中提到，如果待应用的 Entry 记录中封装了 CompactionRequest，会调用 applierV3backend.Compact()方法处理，在其中会将压缩操作放入 FIFO 调度器中执行。该方法还会返回一个通道，当压缩操作真正完成之后，会关闭该通道实现通知的效果。

回顾完 applierV3 接口中压缩相关的操作之后，下面看一下 EtcdServer.Compact()方法的具体实现：

```
    func (s *EtcdServer) Compact(ctx context.Context, r *pb.CompactionRequest)
        (*pb.CompactionResponse, error) {
```

```
    // 将 CompactionRequest 请求封装成 MsgProp 消息并发送出去，并等待该其中的 Entry 被应用
    result, err := s.processInternalRaftRequestOnce(ctx, pb.InternalRaftRequest{
        Compaction: r})
    if r.Physical && result != nil && result.physc != nil {
      // 等待压缩操作结束。上面提到的 applierV3backend.Compact()方法返回的通道就
      // 是这里的 result.physc 通道
      <-result.physc
      s.be.ForceCommit() // 提交压缩操作使用的事务
    }
    // 错误处理(略)
    resp.Header.Revision = s.kv.Rev()
    return resp, nil
}
```

5. 启动服务

介绍完 EtcdServer 对 RaftKV 接口的实现之后，我们看一下服务端的启动流程，其中就包含了启动 GRPC 服务端的相关代码。

在 etcd.go 文件中的 StartEtcd()函数负责启动 etcd 的服务端，每个节点都会对外提供两组 URL 地址，一组是与集群中其他节点交互的 URL 地址（Peer URL），另一组是与客户端交互的 URL 地址（Client URL）。StartEtcd()函数首先会为每个 URL 地址创建相应的 Listener 实例并记录到指定的字段中，然后调用前文介绍的 etcdserver.NewServer()函数创建 EtcdServer 实例，之后调用 EtcdServer.Start()方法启动该实例，最后调用 Etcd.serve()方法对外提供服务。StartEtcd()函数的具体实现如下所示。

```
func StartEtcd(inCfg *Config) (e *Etcd, err error) {
    // 检测配置(略)
    serving := false  // 标识是否正在提供服务
    e = &Etcd{cfg: *inCfg, stopc: make(chan struct{})} // 创建 Etcd
    // 初始化 Etcd.Peers 字段，其中为每个 Peer URL 创建相应的 Listener 实例，错误处理(略)
    if e.Peers, err = startPeerListeners(cfg); ...
    // 初始化 Etcd.sctxs 字段，其中为每个 Client URL 创建相应的 Listener 实例，错误处理(略)
    if e.sctxs, err = startClientListeners(cfg); ...
    for _, sctx := range e.sctxs { // 将 sctxs 字段中记录的 Listener 实例添加到 Clients 字段中
      e.Clients = append(e.Clients, sctx.l)
    }

    // 初始化集群中其他节点地址和自身的 token(略)
```

```
    srvcfg := &etcdserver.ServerConfig{ // 创建 ServerConfig(略) }
    if e.Server, err = etcdserver.NewServer(srvcfg); ... // 创建 EtcdServer 实例，错误
                                                          // 处理(略)
    // 创建 PeerHandler，NewPeerHandler()函数的具体首先在上一章中已经介绍过了
    ph := etcdhttp.NewPeerHandler(e.Server)
    for _, p := range e.Peers { // 使用 Etcd.Peers 中记录的 Listener，创建相应的 http.Server 实例
        srv := &http.Server{
            Handler:     ph,
            ReadTimeout: 5 * time.Minute,
        }
        l := p.Listener
        p.serve = func() error { return srv.Serve(l) } // 启动 http.Server 实例的回调函数
        p.close = ...  // 关闭 http.Server 实例的回调函数
    }
    e.Server.Start() // 启动 EtcdServer 实例，该过程在上一章中已经详细分析过了，这里不再赘述
    if err = e.serve();...  // 启动
    serving = true
    return
}
```

接下来介绍 startPeerListeners()函数，该函数中会为每个 Peer URL 创建相应的 Listener 实例。后面会使用这些 Listener 创建相应的 http.Server 实例，从而监听 Peer URL 上的请求。startPeerListeners()函数的具体实现如下：

```
func startPeerListeners(cfg *Config) (peers []*peerListener, err error) {
    // 在 Config.LPUrls 中记录了当前节点与集群中其他节点交互的 URL 地址
    peers = make([]*peerListener, len(cfg.LPUrls))
    for i, u := range cfg.LPUrls {  // 为 LPUrls 中的每个 URL 地址都创建相应的 Listener
        peers[i] = &peerListener{close: func(context.Context) error { return nil }}
        peers[i].Listener, err = rafthttp.NewListener(u, &cfg.PeerTLSInfo)
        peers[i].close = func(context.Context) error {
            return peers[i].Listener.Close()
        }
    }
    return peers, nil // 返回 peers
}
```

下面看一下 startClientListeners()函数，其中会为每个 Client URL 地址创建相应的 serverCtx

实例。在 serverCtx 实例中记录了对应的 Client URL、相应的 Listener 实例和用户自定义 Handler 等信息。startClientListeners()函数的具体实现如下所示。

```
func startClientListeners(cfg *Config) (sctxs map[string]*serveCtx, err error) {
  // 该 map 用来记录 Client URL 与对应 serveCtx 实例的对应关系
  sctxs = make(map[string]*serveCtx)
  for _, u := range cfg.LCUrls {
    sctx := newServeCtx()  // 每个 Client Url 都对应一个 serveCtx 实例
    proto := "tcp"
    addr := u.Host
    // 调整 proto 和 addr(略)
    if sctx.l, err = net.Listen(proto, addr); ... // 创建 Listener, 错误处理(略)
    sctx.addr = addr // 更新 addr 字段
    // 根据不同的系统和协议, 对 sctx.l 中记录的 Listener 进行调整(略)
    for k := range cfg.UserHandlers { // 设置用户自定义的 Handler
      sctx.userHandlers[k] = cfg.UserHandlers[k]
    }
    sctx.serviceRegister = cfg.ServiceRegister
    sctxs[addr] = sctx // 记录当前 serverCtx 实例
  }
  return sctxs, nil
}
```

最后是 Etcd.serve()方法，该方法中会启动 Peer URL 对应的 http.Server 实例，开始监听来自集群中其他节点的请求，另外还会启动 GRPC 服务，开始接收客户端的请求，该方法的具体实现如下所示。

```
func (e *Etcd) serve() (err error) {
  // 省略相关检测
  for _, pl := range e.Peers {
    // 每个 Peer URL 对应的 http.Server 实例都会在单独的 goroutine 中启动
    go func(l *peerListener) {
      e.errHandler(l.serve())  // 调用前面介绍的 serve()回调函数, 启动 http.Server 实例
    }(pl)
  }

  // 初始化 v2 版本 API 使用的 Handler 实例
  var h http.Handler
```

```
    if e.Config().EnableV2 {
        // 注册完整的 v2 版本的 Handler，可以正常响应 Client v2 的请求
        h = v2http.NewClientHandler(e.Server, e.Server.Cfg.ReqTimeout())
    } else {
        mux := http.NewServeMux()
        etcdhttp.HandleBasic(mux, e.Server) // 只提供基本的查询功能，不响应 Client v2 的请求
        h = mux
    }
    h = http.Handler(&cors.CORSHandler{Handler: h, Info: e.cfg.CorsInfo})
    // 每个 Client URL 对应的服务都会在单独的 goroutine 中启动
    for _, sctx := range e.sctxs {
        go func(s *serveCtx) {
            // 这里调用 serveCtx.serve()方法完成启动
            e.errHandler(s.serve(e.Server, ctlscfg, h, e.errHandler))
        }(sctx)
    }
    return nil
}
```

下面我们进入 serveCtx.serve()方法进行进一步分析。在上一章介绍 EtcdServer 时提到，当前其完全初始化之后会关闭 EtcdServer.readych 通道，serveCtx.serve()方法在开始处就会阻塞监听该通道是否关闭，从而决定是否继续对外提供服务。serveCtx.serve()方法的具体实现如下所示。

```
func (sctx *serveCtx) serve(s *etcdserver.EtcdServer, tlscfg *tls.Config,
    handler http.Handler, errHandler func(error)) error {
    <-s.ReadyNotify() // 阻塞监听 EtcdServer.readych 通道
    m := cmux.New(sctx.l)
    v3c := v3client.New(s)
    servElection := v3election.NewElectionServer(v3c)
    servLock := v3lock.NewLockServer(v3c)

    if sctx.insecure {
        gs := v3rpc.Server(s, nil) // 在其中会完成 GRPC 服务的注册
        sctx.grpcServerC <- gs // 将注册之后的 grpc.Server 写入该通道中
        v3electionpb.RegisterElectionServer(gs, servElection)
        v3lockpb.RegisterLockServer(gs, servLock)
        grpcl := m.Match(cmux.HTTP2())
```

```
        go func() { errHandler(gs.Serve(grpcl)) }() // 在单独后台的 goroutine 中启动 GRPC 服务

        httpmux := sctx.createMux(gwmux, handler)
        srvhttp := &http.Server{
          Handler: httpmux,
        }
        httpl := m.Match(cmux.HTTP1())
        // 在单独的后台 goroutine 中启动一个 http.Server 实例，用来处理 Client V2 请求
        go func() { errHandler(srvhttp.Serve(httpl)) }()
      }
      if sctx.insecure {
        // 加密场景下的初始化与上述初始化类似，代码不再粘贴出来。需要注意的是，在这里注册之后
        // 得到的 grpc.Server 实例也会写入相应的 grpcServerC 通道中，在该节点关闭时，会在
        // Etcd.Close()方法中关闭这些 GRPC 服务
      }
      close(sctx.grpcServerC)
      return m.Serve()
    }
```

最后简单介绍一下 v3rpc.Server()函数，这里是完成 gRPC 服务的注册的地方，在该函数中不仅完成了 KVServer 服务的注册，还完成了 WatcherServer 和 LeaseServer 多个其他服务的注册，其具体实现如下：

```
func Server(s *etcdserver.EtcdServer, tls *tls.Config) *grpc.Server {
  var opts []grpc.ServerOption
  opts = append(opts, grpc.CustomCodec(&codec{}))
  opts = append(opts, grpc.UnaryInterceptor(newUnaryInterceptor(s)))
  opts = append(opts, grpc.StreamInterceptor(newStreamInterceptor(s)))
  opts = append(opts, grpc.MaxConcurrentStreams(maxStreams))
  grpcServer := grpc.NewServer(opts...)

  pb.RegisterKVServer(grpcServer, NewQuotaKVServer(s)) // 注册 KVServer
  pb.RegisterWatchServer(grpcServer, NewWatchServer(s)) // 注册 WatchServer
  // 注册 LeaseServer、ClusterServer、AuthServer、MaintenanceServer 等服务(略)
  return grpcServer
}
```

8.2.3　Client

在本节前面的内容中，我们介绍了 protoc 生成的服务端代码，以及我们如何使用这些生成的代码搭建一个 GRPC 服务。在此我们将来介绍 protoc 生成的客户端代码，以及如何使用这些代码搭建一个客户端。

首先，在上一节介绍的 rpc.pb.go 文件中会为客户端生一个名为 KVClient 的接口，同时也会生成该接口的实现——结构体 kVClient，如图 8-3 所示。在 etcd-clientv3 中还有提供了另外两个 KVClient 接口的实现，在后面我们会一一介绍。

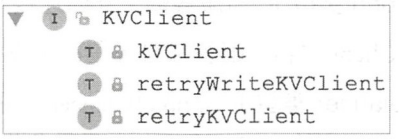


图 8-3

etcd V3 版本的客户端中的核心结构体是 Client，在其中嵌套了多个接口，这里简单介绍每个接口的含义。

- **KV 接口**：负责处理键值对操作，例如，增删改查键值对数据、触发压缩操作等。
- **Cluster 接口**：负责完成集群节点的管理操作，例如，增删改查集群中的节点信息。
- **Lease 接口**：负责租约相关的操作，例如，新建和回收租约、为租约续期等。
- **Watcher 接口**：负责为指定的 Key 添加 Watcher。
- **Auth 接口**：负责权限相关的操作，例如，添加用户或角色、为用户分配角色等。
- **Maintenance 接口**：提供了获取 Alarm 列表、关闭 Alarm 的功能，该接口还提供了读取指定节点状态的功能、触发指定节点进行压缩的功能，以及读取指定节点快照的功能。

在本节中，我们重点以 KV 接口为例进行介绍，其具体定义如下：

```
type KV interface {
    // 向集群写入指定的键值对
    Put(ctx context.Context, key, val string, opts ...OpOption) (*PutResponse, error)
    // 查询指定键值对
    Get(ctx context.Context, key string, opts ...OpOption) (*GetResponse, error)
    // 删除指定的键值对
    Delete(ctx context.Context, key string, opts ...OpOption) (*DeleteResponse, error)
    // 触发压缩操作
```

仅供非商业用途或交流学习使用


```
    Compact(ctx context.Context, rev int64, opts ...CompactOption)
        (*CompactResponse, error)
    // 应用单个操作，上面的 Put()、Get()等方法都是调用该方法实现的
    Do(ctx context.Context, op Op) (OpResponse, error)
    // 开启一个事务
    Txn(ctx context.Context) Txn
}
```

结构体 Client 中其他关键字段的具体含义如下所示。

- conn（*grpc.ClientConn 类型）：与服务端交互的连接，在前面使用过，不再赘述。
- cfg（Config 类型）：Client 的相关配置都封装在该字段中。
- balancer（*simpleBalancer 类型）：simpleBalancer 继承了 GRPC 中的 Balancer 接口，它会选择合适的地址并相应的 gRPC 连接，实现了负载均衡的功能。
- retryWrapper（retryRpcFunc 类型）：retryRpcFunc 主要负责重试。
- Username 和 Password（string 类型）：当前客户端的用户名和密码，当客户端正式发起请求之前，会先通过用户名和密码检测相应权限。

1. 初始化

Client 实例的初始化过程是在 newClient()函数中完成的，该方法首先会创建 Client 实例，然后创建 grpc.ClientConn 实例，最后初始化前面介绍的 GRPC 服务的客户端。newClient()函数的具体实现如下所示。

```
func newClient(cfg *Config) (*Client, error) {
    // 检测配置信息(略)
    client := &Client{ // 创建 Client 实例，注意，此时的 conn 字段还未初始化
        conn:     nil,
        dialerrc: make(chan error, 1),
        cfg:      *cfg,
        creds:    creds,
        ctx:      ctx,
        cancel:   cancel,
    }
    if cfg.Username != "" && cfg.Password != "" { // 记录配置信息中携带的用户名和密码
        client.Username = cfg.Username
        client.Password = cfg.Password
    }
```

仅供非商业用途或交流学习使用


```
    // 创建 simpleBalancer 实例并初始化 Client.balancer 字段
    client.balancer = newSimpleBalancer(cfg.Endpoints)
    // 建立网络连接，在后面会详细介绍该方法建立连接的过程，错误处理(略)
    conn, err := client.dial(cfg.Endpoints[0], grpc.WithBalancer(client.balancer))
    client.conn = conn // 初始化 Client.conn 字段
    // 初始化 Client.retryWrapper 字段和 retryAuthWrapper 字段
    client.retryWrapper = client.newRetryWrapper()
    client.retryAuthWrapper = client.newAuthRetryWrapper()

    if cfg.DialTimeout > 0 { // 阻塞等待连接建立完成
      hasConn := false
      waitc := time.After(cfg.DialTimeout)
      select {
      case <-client.balancer.readyc: // 连接建立完成时，会从该通道中收到信号
        hasConn = true
      case <-waitc: // 超时
      }
      if !hasConn { // 超时之后连接未建立，则关闭连接并返回 ErrClientConnTimeout 错误
        ... ...
      }
    }
    // 初始化前面介绍的多个 gRPC 服务的客户端
    client.Cluster = NewCluster(client)
    client.KV = NewKV(client)
    client.Lease = NewLease(client)
    client.Watcher = NewWatcher(client)
    client.Auth = NewAuth(client)
    client.Maintenance = NewMaintenance(client)
    // 在这里启动一个后台 goroutine，定时同步集群中各个节点暴露的 URL
    go client.autoSync()
    return client, nil
}
```

接下来我们分析 Client.dial()方法是如何建立网络连接的，其具体实现如下：

```
func (c *Client) dial(endpoint string, dopts ...grpc.DialOption) (*grpc.ClientConn,
error) {
    opts := c.dialSetupOpts(endpoint, dopts...)  // 创建连接使用的参数信息
```

仅供非商业用途或交流学习使用


```
    host := getHost(endpoint) // 需要连接的 host 地址
    if c.Username != "" && c.Password != "" {
        // 设置用户名和密码(略)
    }
    opts = append(opts, c.cfg.DialOptions...)
    // 创建连接，错误处理(略)
    conn, err := grpc.DialContext(c.ctx, host, opts...)
    return conn, nil
}
```

最后，我们来看一下 newClient()函数中启动的后台 goroutine，该 goroutine 执行的是 Client.autoSync()方法，具体实现如下：

```
func (c *Client) autoSync() {
    // 检测是否开启了自动同步(略)
    for {
        select {
        case <-time.After(c.cfg.AutoSyncInterval): // 间隔一段时间，进行一次同步
            ctx, _ := context.WithTimeout(c.ctx, 5*time.Second)
            // 调用 Sync()方法进行同步
            if err := c.Sync(ctx); err != nil && err != c.ctx.Err()...
        }
    }
}
```

在 Client.Sync()方法中会向集群请求当前的节点列表，并更新本地的缓存，其具体实现如下所示。

```
func (c *Client) Sync(ctx context.Context) error {
    mresp, err := c.MemberList(ctx) // 通过 GRPC 请求集群的节点列表
    if err != nil {
        return err
    }
    var eps []string
    for _, m := range mresp.Members { // 遍历集群返回的节点列表
        eps = append(eps, m.ClientURLs...)
    }
    c.SetEndpoints(eps...) // 更新 Client 记录的 URL 集合
```

仅供非商业用途或交流学习使用


```
    return nil
}
```

2. kv

前面提到，Client 中内嵌了 KV 接口，在 Client 实例的初始化过程可以看到，其中由 NewKV() 函数创建了 kv 实例。在 kv 中只有一个 remote 字段，该字段实际上指向一个 RetryKVClient 实例（RetryKVClient 实现了 KV 接口）。

结构体 kv 中的 Put()、Get()和 Delete()三个方法的实现比较简单，直接调用 kv.Do()方法实现。在 kv.Do()方法的实现中会循环调用 kv.do()方法，具体实现如下所示。

```
func (kv *kv) Do(ctx context.Context, op Op) (OpResponse, error) {
    for {
        resp, err := kv.do(ctx, op)
        // 如果此次操作过程中未发生错误，或发生了不可重试的错误，则直接返回(略)
        if op.isWrite() { // 如果此次操包含写操作，则不会重试，而是返回错误信息
            return resp, toErr(ctx, err)
        }
    }
}
```

在 kv.do()方法中会根据此次操作的类型，分别调用 KVClient 相应的方法向集群发送请求，具体实现如下所示。

```
func (kv *kv) do(ctx context.Context, op Op) (OpResponse, error) {
    var err error
    switch op.t {
    case tRange: // 查询操作，调用 KVClient.Range()方法进行处理
        var resp *pb.RangeResponse
        resp, err = kv.remote.Range(ctx, op.toRangeRequest(), grpc.FailFast(false))
        return OpResponse{get: (*GetResponse)(resp)}, nil
    case tPut: // 写入操作，则调用 KVClient.Put()方法进行处理
        var resp *pb.PutResponse
        r := &pb.PutRequest{Key: op.key, Value: op.val, Lease: int64(op.leaseID),
            PrevKv: op.prevKV, IgnoreValue: op.ignoreValue, IgnoreLease: op.ignoreLease}
        resp, err = kv.remote.Put(ctx, r)
        return OpResponse{put: (*PutResponse)(resp)}, nil
    case tDeleteRange: // 删除操作，调用 KVClient.DeleteRange()方法进行处理
        var resp *pb.DeleteRangeResponse
```

仅供非商业用途或交流学习使用


```
        r := &pb.DeleteRangeRequest{Key: op.key, RangeEnd: op.end, PrevKv: op.prevKV}
        resp, err = kv.remote.DeleteRange(ctx, r)
        return OpResponse{del: (*DeleteResponse)(resp)}, nil
    default: panic("Unknown op")
    }
    return OpResponse{}, err
}
```

kv.Compact()方法的实现比较简单，是直接调用了 KVClient.Compact()方法实现的，这里不再赘述。kv.Txn()方法的实现也比较简单，直接返回了一个 txn 实例，在 txn 实例中封装了多个操作。结构体 txn 实现了 Txn 接口，Txn 接口的定义如下：

```
type Txn interface {
    If(cs ...Cmp) Txn // 添加 Compare 条件
    Then(ops ...Op) Txn // 如果 Compare 条件全部成立，则执行 Then()方法中添加的全部操作
    Else(ops ...Op) Txn // 如果 Compare 条件不成立，则执行 Else()方法中添加的全部操作
    Commit() (*TxnResponse, error) // 提交 Txn
}
```

接下来看一下结构体 txn 中的各个字段的含义。

- kv（*kv 类型）：关联的 kv 实例。
- cmps（[]*pb.Compare 类型）：用于记录 Compare 条件的集合。
- sus（[]*pb.RequestOp 类型）：全部 Compare 条件通过之后执行的操作列表。
- fas（[]*pb.RequestOp 类型）：任一 Compare 条件检测失败之后执行的操作列表。
- cif（bool 类型）：是否设置 Compare 条件，设置一次则该字段就会被设置为 true。
- cthen（bool 类型）：是否向上面的 sus 集合中添加了操作。
- celse（bool 类型）：是否向上面的 fas 集合中添加操作。
- isWrite（bool 类型）：如果当前 txn 实例中记录的操作全部是读操作，则该字段为 false，否则为 true。

txn.If()、Then()和 Else()方法会将操作添加到指定的集合中，其中 If()方法会将 Compare 条件添加到 cmps 集合中，Then()方法会将操作添加到 sus 集合中并同时更新 isWrite 字段，Else()方法会将操作添加到 fas 集合中并同时更新 isWrite 字段。

这里重点看一下 txn.Commit()方法，该方法最终会调用 KVClient.Txn()方法进行处理，具体实现如下所示。

仅供非商业用途或交流学习使用


```
func (txn *txn) Commit() (*TxnResponse, error) {
    txn.mu.Lock()
    defer txn.mu.Unlock()
    for {
        resp, err := txn.commit()
        // 如果此次操作过程中未发生错误，或发生了不可重试的错误，则直接返回(略)
        if txn.isWrite { // 如果此次操包含写操作，则不会重试，而是返回错误信息
            return nil, toErr(txn.ctx, err)
        }
    }
}
// 下面是 txn.commit()方法的具体实现
func (txn *txn) commit() (*TxnResponse, error) {
    // 创建 TxnRequest 实例
    r := &pb.TxnRequest{Compare: txn.cmps, Success: txn.sus, Failure: txn.fas}
    var opts []grpc.CallOption
    if !txn.isWrite { // 根据 isWrite 字段设置相应请求参数
        opts = []grpc.CallOption{grpc.FailFast(false)}
    }
    // 调用 KVClient.Txn()方法与服务端交互
    resp, err := txn.kv.remote.Txn(txn.ctx, r, opts...)
    return (*TxnResponse)(resp), nil
}
```

3. RetryKVClient

在上一小节的介绍中提到，kv.remote 字段实际上指向一个 RetryKVClient 实例。结构体 RetryKVClient 中内嵌了 retryWriteKVClient，而在结构体 retryWriteKVClient 中内嵌了 KVClient 接口并且封装了一个 retryRpcFunc 回调函数（retryf 字段）。

在 RetryKVClient 实例初始化时，会将 retryRpcFunc 字段指向 Client.retryWrapper 字段，而 Client.retryWrapper 字段指向的回调函数则是在 newRetryWrapper()中定义的，具体实现如下所示。

```
func (c *Client) newRetryWrapper() retryRpcFunc {
    return func(rpcCtx context.Context, f rpcFunc) error {
        for {
            err := f(rpcCtx) // 调用 rpcFunc 函数
            // 如果未出现错误，则返回
```

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用


```
            // 如果出现错误，则对错误码进行分析，只有出现 Unavailable 错误码时，才会重试(略)
        }
    }
}
```

在 retryWriteKVClient 的实现中，都是通过上述回调函数完成 RPC 调用的，这样就可以实现重试的功能，这里以 Put()方法为例进行介绍：

```
func (rkv *retryWriteKVClient) Put(ctx context.Context, in *pb.PutRequest,
     opts ...grpc.CallOption) (resp *pb.PutResponse, err error) {
  err = rkv.retryf(ctx, func(rctx context.Context) error { // 调用 retryf()回调函数
    // 调用 KVClient.Put()方法发送 RPC 请求，该 KVCleint 实例就是前面通过 protoc 生成的客户端代码
    resp, err = rkv.KVClient.Put(rctx, in, opts...)
    return err
  })
  return resp, err
}
```

retryWriteKVClient 中实现的其他方法与 Put()方法类似，这里不再赘述，感兴趣的读者可以分析相应代码。

到目前为止，Client v3 中 KV 服务的整个运作流程，以及涉及的具体实现就全部介绍完了。除了 KV 服务的其他服务，例如，Cluster、Auth、Lease、Maintenance 和 Watch 等，与 KV 服务的运转流程很类似，只不过是服务端底层调用了不同的组件，支持了不同方面的功能而已，这里不再一一展开介绍。在了解了 KV 服务的整个流程之后，读者可以很轻松地完成其他服务的代码分析。

8.3 Client v2

etcd v2 版本的 Client 的代码位于 client 包中，它是通过 HTTP+JSON 方式实现的。在上一节介绍 etcd 服务端启动的过程中，我们可以也看到其中专门添加了一个 Handler 用于 Client v2 请求，读者可以回顾上一章对 etcd 服务端的介绍。

8.3.1 KeysAPI接口

我们首先需要了解的是 KeysAPI 接口，该接口中定义了 Client v2 对键值对操作的方法，其具体定义如下所示。

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用


```
type KeysAPI interface {
    // 查询键值对数据
    Get(ctx context.Context, key string, opts *GetOptions) (*Response, error)
    // 添加新的键值对数据
    Set(ctx context.Context, key, value string, opts *SetOptions) (*Response, error)
    Update(ctx context.Context, key, value string) (*Response, error)
    Create(ctx context.Context, key, value string) (*Response, error)
    CreateInOrder(ctx context.Context, dir, value string, opts *CreateInOrderOptions)
        (*Response, error)
    // 删除指定的键值对数据
    Delete(ctx context.Context, key string, opts *DeleteOptions) (*Response, error)
    // 添加 Watcher
    Watcher(key string, opts *WatcherOptions) Watcher
}
```

结构体 httpKeysAPI 是 KeysAPI 接口的实现，其中封装了 httpClient 实例（client 字段），该 httpKeysAPI 中所有方法的实现都是通过委托给 httpClient.Do() 方法实现的。这里以 httpKeysAPI.Set()方法为例进行介绍，其具体实现如下：

```
    func (k *httpKeysAPI) Set(ctx context.Context, key, val string, opts *SetOptions)
(*Response, error) {
        act := &setAction{ // 创建 setAction
            Prefix: k.prefix, // 前缀路径
            Key:    key,
            Value:  val,
        }

        if opts != nil { // 在 setAction 中设置相应参数
            act.PrevValue = opts.PrevValue
            act.PrevIndex = opts.PrevIndex
            act.PrevExist = opts.PrevExist
            act.TTL = opts.TTL
            act.Refresh = opts.Refresh
            act.Dir = opts.Dir
            act.NoValueOnSuccess = opts.NoValueOnSuccess
        }
        doCtx := ctx
```

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用


```
    if act.PrevExist == PrevNoExist {
        doCtx = context.WithValue(doCtx, &oneShotCtxValue, &oneShotCtxValue)
    }
    // 调用 httpClient.Do()方法，错误处理(略)
    resp, body, err := k.client.Do(doCtx, act)
    return unmarshalHTTPResponse(resp.StatusCode, resp.Header, body) // 返回响应信息
}
```

httpKeysAPI 其他方法与 Set()方法类似，这里不再一一展开介绍。

8.3.2 httpClient接口

httpClient 接口中只定义了一个 Do()方法，正如上一小节所示，所有的请求都会经过该方法处理。httpClient 接口的相关实现如图 8-4 所示。

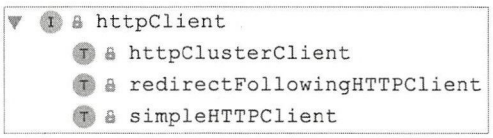


图 8-4

1. httpClusterClient

首先分析结构体 httpClusterClient，httpKeysAPI.client 字段实际上也指向 httpClusterClient 实例。httpClusterClient 中各个字段的含义如下所示。

- clientFactory（httpClientFactory 类型）：工厂函数，用于创建底层的 httpClient 实例。
- endpoints（[]url.URL 类型）：记录集群中所有节点的暴露给客户端的 URL 地址。
- pinned（int 类型）：用于选择重试的 URL 地址。
- rand（*rand.Rand 类型）：随机数，用于选择重试的 URL 地址。
- selectionMode（EndpointSelectionMode 类型）：更新 endpoints 字段的模式，目前有两个可选值，分别是 EndpointSelectionRandom 和 EndpointSelectionPrioritizeLeader。

下面看一下 httpClusterClient.Do()方法，其中会从 endpoints 字段中选择合适的 URL 地址并建立连接，然后发送相应的请求，具体实现如下所示。

```
func (c *httpClusterClient) Do(ctx context.Context, act httpAction)
    (*http.Response, []byte, error) {
    action := act
```

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用


```
    leps := len(c.endpoints)
    // 复制目前集群提供的 URL 地址
    eps := make([]url.URL, leps)
    n := copy(eps, c.endpoints)
    pinned := c.pinned
    var resp *http.Response
    var body []byte

    for i := pinned; i < leps+pinned; i++ {
        k := i % leps
        hc := c.clientFactory(eps[k]) // 从 eps 中选择一个 URL 地址创建连接
        resp, body, err = hc.Do(ctx, action) // 调用 httpClient.Do()方法向服务端发送请求
        if err != nil {
            // 错误处理(略)
        } else if resp.StatusCode/100 == 5 {
            // 处理 5XX 的错误码
        }

        if k != pinned {
            c.pinned = k
        }
        return resp, body, nil
    }
    return nil, nil, cerr
}
```

在 Client v2 中也有与 Client v3 类似的同步机制，其主要实现在 httpClusterClient.Sync()方法中。Sync()方法首先会请求当前集群中所有节点提供的 ClientURL 地址，然后根据当前 selectionMode 字段指定的模式修改 pinned 字段值，最后更新 endpoints 字段中记录的 ClientURL 地址。Sync()方法发具体实现如下所示。

```
func (c *httpClusterClient) Sync(ctx context.Context) error {
    mAPI := NewMembersAPI(c)
    ms, err := mAPI.List(ctx) // 请求集群中各个节点的信息
    var eps []string
    for _, m := range ms {  // 将获取到的 URL 地址添加到 eps 数组中
        eps = append(eps, m.ClientURLs...)
    }
```

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用


```
    neps, err := c.parseEndpoints(eps) // 解析 URL，错误处理(略)
    npin := 0
    switch c.selectionMode {
    case EndpointSelectionRandom:
      eq := endpointsEqual(c.endpoints, neps) // 比较 ClientURL 地址是否变化
      if eq {
        return nil
      }
      neps = shuffleEndpoints(c.rand, neps) // 洗牌，打乱 URL 顺序
    case EndpointSelectionPrioritizeLeader:
      nle, err := c.getLeaderEndpoint(ctx, neps) // 获取 Leader 提供的 ClientURL，错误处理(略)
      for i, n := range neps { // 遍历全部的 URL
        if n.String() == nle {
          npin = i // 将 npin 指向 Leader 提供的 ClientURL 地址的下标
          break
        }
      }
    default: // 返回错误(略)
    }
    c.endpoints = neps // 更新 endpoints 字段
    c.pinned = npin // 更新 pinned 字段
    return nil
}
```

httpClusterClient.clientFactory 字段指向了一个 newHTTPClientFactory()工厂函数，该函数负责根据指定的 ClientURL 创建 httprClient 实例，具体实现如下：

```
func newHTTPClientFactory(tr CancelableTransport, cr CheckRedirectFunc,
      headerTimeout time.Duration) httpClientFactory {
    return func(ep url.URL) httpClient { // 工厂函数
      return &redirectFollowingHTTPClient{ // 创建 redirectFollowingHTTPClient 实例
        checkRedirect: cr,
        client: &simpleHTTPClient{
          transport:     tr,
          endpoint:      ep,
          headerTimeout: headerTimeout,
        },
      }
    }
}
```

仅供非商业用途或交流学习使用

2. redirectFollowingHTTPClient

httpClusterClient.clientFactory 工厂函数返回的是 redirectFollowingHTTPClient 实例，redirectFollowingHTTPClient 也实现了 httpClient 接口，其 Do()方法主要处理 Redirect 跳转的响应，具体实现如下：

```
func (r *redirectFollowingHTTPClient) Do(ctx context.Context, act httpAction)
(*http.Response, []byte, error) {
    next := act
    for i := 0; i < 100; i++ {
        // 检测当前 Redirect 的次数，是否还能继续 Redirect(略)
        resp, body, err := r.client.Do(ctx, next)
        if err != nil {
            return nil, nil, err
        }
        if resp.StatusCode/100 == 3 { // 处理 3XX 的响应
            hdr := resp.Header.Get("Location") // 获取响应的 Location，错误处理(略)
            loc, err := url.Parse(hdr) // 解析 Location 指定的地址
            if err != nil {
                return nil, nil, fmt.Errorf("Location header not valid URL: %s", hdr)
            }
            next = &redirectedHTTPAction{ // 重新创建 redirectedHTTPAction 实例，并重试
                action:  act,
                location: *loc,
            }
            continue
        }
        return resp, body, nil
    }
    return nil, nil, errTooManyRedirectChecks
}
```

3. simpleHTTPClient

在前面介绍的 newHTTPClientFactory()工厂函数中可以看到，redirectFollowingHTTP-Client.client 字段指向了一个 simpleHTTPClient 实例，结构体 simpleHTTPClient 也是 httpClient 接口的实现之一。simpleHTTPClient 是功能最简单，也是最基础的 httpClient 接口实现。simpleHTTPClient 中各个字段的含义如下所示。

- transport (CancelableTransport 类型)：内嵌了 http.RoundTripper，用于发送 HTTP

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

请求并获取相应的响应。

- endpoint（url.URL 类型）：请求的 URL 地址。
- headerTimeout（time.Duration 类型）：请求的超时时间。

接下来看一下 simpleHTTPClient.Do()方法的具体实现如下：

```
func (c *simpleHTTPClient) Do(ctx context.Context, act httpAction) (*http.Response,
[]byte, error) {
    req := act.HTTPRequest(c.endpoint) // 创建 Request 实例，错误处理(略)
    // 设置超时时间(略)
    rtchan := make(chan roundTripResponse, 1)
    go func() { // 启动一个后台 goroutine 发送请求
      resp, err := c.transport.RoundTrip(req)
      // 将响应写入 rtchan 通道中
      rtchan <- roundTripResponse{resp: resp, err: err}
      close(rtchan)
    }()
    var resp *http.Response
    var err error

    select {
    case rtresp := <-rtchan: // 监听 rtchan 通道获取响应
      resp, err = rtresp.resp, rtresp.err
    // 其他情况(略)
    }
    var body []byte
    done := make(chan struct{})
    go func() { // 启动一个后台 goroutine 读取响应的数据
      body, err = ioutil.ReadAll(resp.Body)
      done <- struct{}{} // 读取完成后，向 done 通道发送信号
    }()
    select {
    case <-done: // 阻塞等待后台 goroutine 读取响应体结束
    // 其他情况(略)
    }
    return resp, body, err
}
```

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

除了 KeysAPI，Client v2 中其他的服务，例如，MembersAPI 等，底层也是依赖 httpClient 接口向服务端发送 HTTP 请求并获取响应的，这里不再一一展开介绍。

本章小结

本章主要介绍了 etcd 客户端的具体实现，其中包括 v3 和 v2 两个版本的客户端实现。Client v3 的具体实现在 client v3 包中，Client v3 是通过 GRPC 的方式与服务端进行交互的。在本章开始，我们介绍了 GRPC 的基本使用方式，其中包括 proto 文件的定义、protoc 编译器的使用，以及在生成代码的基础上完成客户端和服务端。

随后我们详细分析了 Client v3 中 KV 服务的内容，首先是 proto 文件的服务定义，以及生成的客户端和服务端的接口，然后介绍了服务端的 EtcdServer 结构体对 KV 接口等的实现，最后详细分析了客户端的实现。

在本章的最后一节，我们简单介绍了 Client v2 的大致实现。v2 版本的客户端是通过 HTTP+JSON 的方式与服务端进行通信的。这里我们只是简单介绍了 KeysAPI 接口、底层使用的 httpClient 接口及具体实现，其他的接口留给读者自行分析。

希望通过本章的介绍，读者不仅了解如何使用 etcd 的客户端，还能够深入了解其工作原理。

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

参考文献

参考书籍

[1] Ongaro D, Ousterhout J. In search of an understandable consensus algorithm[J]. Draft of October, 2014.

[2] Brian Ketelsen, Erica Martin, William Kennedy. Go In Action.

[3] Alan A. A. Donovan. Brian W. Kernighan. The Go Programming Language.

参考网络资源

Raft 协议：http://raftconsensus.github.io

etcd 官方文档：https://coreos.com/etcd/

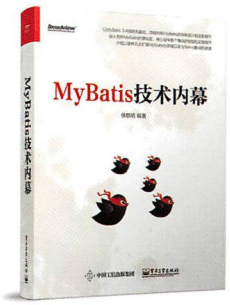
neverchanje 的博客：http://blog.neverchanje.com/

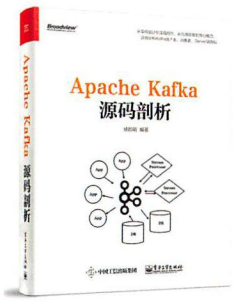
云溪社区：https://yq.aliyun.com/

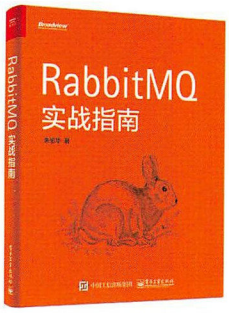
仅供非商业用途或交流学习使用

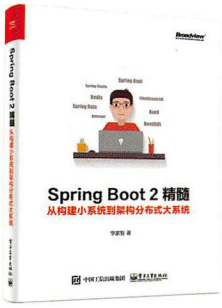
仅供非商业用途或交流学习使用

好书分享













拒绝堆砌臃肿，支持纯正原创

欢迎投稿：chenxm@phei.com.cn

Broadview®
www.broadview.com.cn
博文视点·IT出版旗舰品牌
技术凝聚实力·专业创新出版

etcd技术内幕

本书从源码角度深入剖析etcd，主要内容如下：

- ◆etcd的背景知识，如何搭建源码环境及相关命令
- ◆Raft协议的背景，如何处理各种异常场景及相关扩展
- ◆etcd-raft模块对Raft协议的实现，etcd自带的raftexample示例
- ◆HTTP编程基础，etcd-rafthttp模块的工作原理和具体实现
- ◆etcd中如何处理WAL日志文件及快照数据文件
- ◆etcd的底层存储（etcd v2和etcd v3）
- ◆etcd服务端及扩展

博文视点Broadview
新浪微博 weibo.com
@博文视点Broadview

责任编辑：陈晓猛
封面设计：李　玲

上架建议：计算机 / 分布式存储

ISBN 978-7-121-34386-5
9 787121 343865 >
定价：89.00元